Bull. Nov. Comp. Center, Comp. Science, 35 (2013), 69–83
© 2013 NCC Publisher

Constructions used in associative parallel algorithms for undirected graphs. Part 1

A. S. Nepomniaschaya

Abstract. The paper selects constructions used to represent a group of algorithms for undirected graphs given as a list of edges and their weights on a model of associative (content addressable) parallel systems with vertical processing (the STAR–machine). To this end, the paper analyzes the implementation on the STAR–machine of classical algorithms of Prim–Dijkstra, Kruskal, and Gabow, the method of finding paths with respect to a given spanning tree and its application to solving some tasks. Moreover, the paper proposes an improved version of implementing the Gabow algorithm for finding the smallest spanning tree with a degree constraint of a vertex on the STAR–machine.

Keywords: undirected graph, spanning tree, chord, minimum spanning tree, connected component, fundamental set of circuits, vertical processing system.

1. Introduction

Associative processing is a completely different way of storing, manipulating, and retrieving data as compared to traditional computation techniques. Associative (content addressable) parallel processors of the SIMD type with simple processing elements offer distinctive advantages over other parallel systems such as data parallelism at the base level, the use of 2D tables as the basic data structure, massively parallel search by contents, and processing of unordered data [19]. Associative parallel systems are best suited to solving non-numerical problems such as graph theory, computational geometry, relational database processing, image processing, and genome matching. In [18], the search and data selection algorithms for both bit–serial (vertical) and fully parallel associative processors were described. In [3], the depth search machines and their applications to computational geometry, relational databases, and expert systems were investigated. In [4, 5], an experimental implementation of a multi–comparand multi-search associative processor and some parallel algorithms for search problems in computational geometry were considered. In [6], an associative graph machine (the AG–machine) and its possible hardware implementation were suggested. It performs bit–serial and fully parallel associative processing of matrices representing graphs as well as some basic set operations on matrices (sets of columns).

Of special interest is the class of associative parallel processors of the SIMD type with vertical processing because they simulate the complete associative processing at the micro level. In [9], a model of the SIMD type with vertical processing (the STAR–machine) was proposed. It is useful for specifying and analyzing new associative hardware devices and for designing associative algorithms. Associative parallel algorithms are represented as corresponding procedures for the STAR–machine. In [7], basic associative parallel algorithms were presented used to design different associative algorithms for different applications. Let us enumerate some results of solving graph problems on the STAR–machine. In [9, 10], we construct a natural straight forward implementation on the STAR–machine of the classical graph algorithms of Dijkstra and Bellman–Ford for finding the single–source shortest paths. In [12], we propose associative parallel algorithms for the dynamic edge update of the minimum spanning trees of undirected graphs. In [14], we present the efficient implementation on the STAR–machine of the Italiano algorithms for the dynamic update of the transitive closure of directed graphs. In [15, 16], we propose the efficient implementation on the STAR–machine of the Ramalingam algorithms for the dynamic update of the shortest paths subgraph of a directed graph with a sink.

The goal of this paper is to select the group of constructions used to represent associative algorithms for undirected graphs given on the STAR–machine as a list of edges and their weights. We first select constructions used to design associative versions of the Prim–Dijkstra and Kruskal algorithms for finding the minimum spanning tree [11]. Then we determine a group of constructions to describe the method of finding the paths with respect to the given spanning tree. To describe this method, we use two associative parallel algorithms. Knowing a graph and a spanning tree, the first algorithm constructs the matrix of tree paths. Knowing a graph, the matrix of tree paths, and a chord, the second algorithm determines the positions of edges belonging to the tree path that connects the end-points of the chord. Further we recall a group of algorithms for undirected graphs using this method [13]. In particular, it includes the following three tasks: to verify a minimum spanning tree in undirected graphs, to update a minimum spanning tree, to find a set of fundamental circuits with respect to a particular spanning tree. In [8], we proposed an associative version of the Gabow algorithm for finding the smallest spanning tree with a degree constraint of a vertex. Here, we propose a new efficient associative parallel algorithm for the auxiliary procedure from [8] that determines the positions of edges to replace a given edge. We do this using the matrix of tree paths described above.

2. An associative parallel machine model

In this section, we first recall the main operations of the STAR–machine. The description of the model is given, for example, in [9].

Let us present some elementary operations and a predicate for variables of the type **slice**.[1]

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 i-th 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 $'0'$.

To execute 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. For simplicity, the notation $Y \neq \emptyset$ means 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**.

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

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

COL(i, T) returns its i-th column.

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

Now, we recall a few basic procedures [7] implemented on the STAR–machine. They use a given slice X to indicate with $'1'$ the row positions used in the corresponding procedure. In [7], we have shown that basic procedures take $O(r)$ time each, where r is the number of bit columns in the corresponding matrix.

The procedure MATCH(T, X, w, Z) determines the positions of the rows of the matrix T that coincide with the 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 GREAT(T, X, v, Z) defines the positions of rows of the given matrix T which are greater than the given pattern v written in binary code. It returns the slice Z, where $Z(i) =' 1'$ if and only if ROW(i, T) $> v$ and $X(i) =' 1'$.

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

The procedure MAX(T, X, Z) is defined by analogy with MIN(T, X, Z).

The procedure TMERGE(T, X, F) writes the rows of the matrix T, indicated with bits $'1'$ in the slice X, into the matrix F. Other rows of the matrix F do not change.

[1]For simplicity let us call *slice* any variable of the type **slice**.

The procedure ADDC(T, X, v, F) adds the binary word v to those rows of the matrix T which are selected with bits $'1'$ in the given slice X, and writes down the result into the corresponding rows of the matrix F. Other rows of the matrix F consist of zeroes.

The procedure SUBTV(T, F, Z, R) writes the result of subtracting the matrix F from the matrix T into the matrix R.

3. Preliminaries

Let $G = (V, E)$ be an *undirected weighted graph* with the set of vertices $V = \{1, 2, \dots, n\}$, the set of edges E and the function wt that assigns a weight to every edge. We assume that $|V| = n$ and $|E| = m$.

In the STAR–machine matrix memory, a graph will be represented as association of the matrices *Left*, *Right*, and *Weight*, where each edge (u, v) is matched with the triple $< u, v, wt(u, v) >$.

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

A *spanning tree* $T = (V, E')$ of the given graph G is a connected acyclic subgraph of G, where $E' \subseteq E$. Each edge $e \in E - E'$ is called a *chord* of G with respect to the spanning tree. Adding a chord to a spanning tree creates precisely one circuit.

A *fundamental set of circuits* is a collection of such circuits with respect to a particular spanning tree.

A *minimum spanning tree* (MST) of G is a spanning tree where the sum of weights of the corresponding edges is minimal.

A connected component is a maximal connected subgraph.

4. Finding a minimum spanning tree of an undirected graph

In this section, we propose two constructions used to represent on the STAR–machine the Prim–Dijkstra and Kruskal algorithms for finding a minimum spanning tree of an undirected graph.

The Prim–Dijkstra algorithm builds a minimum spanning tree by growing a subtree T_S. Initially, T_S consists of a single vertex s. The first edge of T_S is chosen as the edge of the minimal weight which is incident to s. The subtree T_S grows by including edges of the minimal weight that have a single vertex belonging to T_S. The process continues until all vertices are included in T_S.

On the STAR–machine, the Prim–Dijkstra algorithm is represented as a procedure MSTPD [11] whose input parameters are matrices $Left$, $Right$, and $Weight$, the binary representation of a vertex s, and the global slice Z that saves the positions of edges belonging to G. This procedure returns the slice $Tree$ that saves the positions of edges included into the minimum spanning tree. In [11], we prove correctness of this procedure and show that on the STAR–machine it takes $O(n \log n)$ time, where n is the number of vertices in the initial graph.

Let us present two constructions used to represent the procedure MSTPD on the STAR–machine.

The first construction uses a variable *node* of the type **word** and the variables $N1$ and $N2$ of the type **slice**. The variable *node* will save the new vertex that will be included into the fragment of T_S at every step. The variables $N1$ and $N2$ will accumulate the positions of the edges that have a single vertex belonging to the fragment of T_S being built. Initially, the variable *node* saves the vertex s, and the slices $N1$ and $N2$ consist of zeroes.

Construction 1. (*Finding the positions of edges forming a cycle.*)

Let the current vertex *node* be included into the current fragment of the minimum spanning tree T_S. Then we first select the positions of edges whose left vertex coincides with *node* and save them in the slice $N1$. Then we select the positions of edges whose right vertex coincides with *node* and save them in the slice $N2$. After that we intersect the slices $N1$ and $N2$ and delete the positions of edges belonging to this intersection from the slice Z.

Really, every edge from the intersection of $N1$ and $N2$ has the following property: it does not belong to the fragment of T_S being built but its end–points belong to T_S. Therefore such edges must be deleted from the global slice Z because they form a cycle.

To obtain the slices $N1$ and $N2$, the basic procedure MATCH is applied to the matrices $Left$ and $Right$.

Construction 2. (*Finding the current vertex for including into T_S.*)

Let an edge from the i-th position of the association of the matrices $Left$, $Right$, and $Weight$ be added to T_S. Then the new value of the variable *node* is determined as follows.

```
if N1(i)='1' then node:=ROW(i,Right)
else node:=ROW(i,Left);
```

Obviously, in Construction 2 one can use the bit `N2(i)` instead of the bit `N1(i)`.

The Kruskal algorithm builds a minimum spanning tree T_S in the following way.

Initially, all the edges of G are sorted out in the increasing order. The first edge of the tree is the first edge in this new array. At any next step, one chooses, among the edges incident to vertices included into the tree, the uppermost edge in the new array that does not form a cycle.

The associative version of the Kruskal algorithm is given as a procedure MST1 whose input parameters are the matrices $Left$, $Right$, and $Weight$, and the number of vertices n. The procedure returns the slice *Result* that saves the positions of edges belonging to the minimum spanning tree. In

[11], we have shown that the procedure MST1 takes $O(n \log n)$ time.

The procedure MST1 uses the variables $N1$ and $N2$ of the type **slice** and the variables $node1$ and $node2$ of the type **word**. The variables $N1$ and $N2$ are used for the same goal as in the procedure MSTPD. The variables $node1$ and $node2$ are used to select the left and the right vertices of the current edge included in T_S.

The procedure MST1 runs as follows. The first edge of T_S is chosen as the edge having the minimal weight in the new array. Every next edge for T_S is selected as the edge of the minimal weight among the edges having a single vertex belonging to the current T_S. To select the edges that are candidates to be included into the tree, Construction 1 is used.

In [11], we have shown that the execution of the Prim–Dijkstra algorithm and the associative version of the Kruskal algorithm are based on the same method allowing one to determine the *positions* of edges whose addition to the growing tree form a cycle.

5. Computation of functions defined on trees

In this section, we select constructions that will be used in the associative algorithm for finding paths with respect to a given spanning tree [13]. We will use the matrix $Code$ whose every i-th row saves the binary representation of the vertex i $(1 \leq i \leq n)$.

In [20], Tarjan suggests a special technique, path compression in balanced trees, for computing functions defined on trees. In [13], we propose a method for finding tree paths in undirected graphs. It is oriented towards the associative update of information and the representation of a graph as a list of triples. The method consists of two associative parallel algorithms. Knowing a graph and a spanning tree, the first algorithm builds the matrix of tree paths $TPaths$, whose every i-th column saves the positions of edges belonging to the tree path from v_1 to v_i. Knowing a graph, the matrix $TPaths$ and a chord σ, the second algorithm determines the positions of edges belonging to the path that connects end–points of σ. On the STAR–machine, the first algorithm is implemented as a procedure MatrixPaths. It uses the following parameters: the matrices $Left$, $Right$, and $Code$, the spanning tree given as a slice $Tree$ and the number of graph vertices n. It returns the matrix $TPaths$. The procedure uses the following idea. Assume we know the positions of edges included into the tree path from v_1 to v_r. Then we construct a tree path from v_1 to such a vertex v_k that is adjacent to v_r, the corresponding edge γ from the spanning tree connects the vertices v_r and v_k, and the path from v_1 to v_k has not been built yet. The tree path from v_1 to v_k is obtained by adding the position of the edge γ to the path from v_1 to v_r.

The procedure MatrixPaths uses, in particular, the variables $node1$ and *node*2 of the type **word**, $N1$ and $N2$ of the type **slice**, and k and j of the type **integer**. We use the variable *node*1 (respectively, *node*2) to save the binary code of the vertex for which the tree path from v_1 has been constructed (respectively, has not been constructed) and the slice $N1$ (respectively, $N2$) to store the positions of the tree edges whose left (respectively, right) vertex has been included in the tree path from v_1.

The procedure MatrixPaths uses the following three constructions.

Construction 3. (*Finding the current and new vertices included into the path being built.*)

Assume at the current iteration we select the uppermost edge (say, γ) written in the i-th position of $N1\, or\, N2$. Then the end–points *node*1 and *node*2 of γ are determined as follows. If the left end–point of γ has been included into the growing fragment of T_S, then its right end–point is the new vertex for T_S. Otherwise, the new vertex is its left end–point.

We perform this by means of the following operator.

```
if N1(i)='1'then
   begin node1:=ROW(i,Left); node2:=ROW(i,Right) end
else begin node1:=ROW(i,Right); node2:=ROW(i,Left) end.
```

Construction 4. (*Finding a path from the root v_1 to the new vertex included into the path being built.*)

Let at the current iteration the edge γ be selected from the i-th position of $N1\, or\, N2$. Let k and j be decimal numbers of the end–points of γ. Let j be the new vertex added to the fragment of T_S. Then the tree path from v_1 to j is obtained by including the position i of the edge γ into the copy of the k-th column of the matrix $TPaths$.

The next construction uses the variables *node*1 and *node*2 of the type **word**.

Construction 5. (*Finding decimal numbers of end–points of an edge.*)

Let an edge (say, γ) be written in the i-th row of association of the matrices $Left$ and $Right$. Then the binary representation of its end–points is written in the i-th row of the matrices $Left$ and $Right$. Let a variable *node*1 (respectively, *node*2) save the i-th row of the matrix $Left$ (respectively, $Right$). Then the corresponding decimal numbers of the end–points of γ are obtained by applying the basic procedure MATCH and the operation FND to the matrix $Code$.

The second associative algorithm is implemented on the STAR–machine as a procedure PathPositions having the following input parameters: the matrices $Left$, $Right$, $Code$, and $TPaths$, and the position of a chord, say σ. It returns a slice Y, where the positions of edges from the tree path connecting end–points of σ are marked by $'1'$. This procedure uses the following idea. Knowing the position i of a chord σ in the graph representation and the matrix $TPaths$, it determines the decimal numbers of the end–points of σ. The resulting slice Y is obtained by means of the operation xor between the bit columns of the matrix $TPaths$ that correspond to the end–points of σ.

The procedure PathPositions uses, in particular, the variables $node1$ and $node2$ of the type **word** and the variables $n1$ and $n2$ of the type **integer**. The variables $node1$ and $node2$ are used for the same purposes as in the procedure MatrixPaths. The variables $n1$ and $n2$ save the decimal representation of $node1$ and $node2$.

The procedure PathPositions uses the following construction.

Construction 6. (*Building a tree path that connects end–points of a chord.*)

Let the matrix $TPaths$ be given. Let a chord σ be written in the i-th row of the graph representation. Let $n1$ and $n2$ be decimal numbers of the end–points of the chord. Then the tree path connecting the end–points of σ is obtained by means of the operation xor between the $n1$-th and $n2$-th bit columns of the matrix $Tpaths$.

Enumerate some applications of the associative parallel algorithm for finding a tree path between any pair of vertices in undirected graphs. In [1], Chin and Houck considered the following problem of the static update of minimum spanning trees. Let T be a minimum spanning tree in a given undirected graph G. For each edge (v, w) of T it is necessary to determine a chord (x, y) by which (v, w) should be replaced to obtain a new minimum spanning tree if the edge (v, w) is deleted from G. The algorithm of Chin and Houck labels each tree edge from T with its replacement. It takes $O(n^2)$ time. In [19], Tarjan suggested an algorithm for the same task which uses the technique of path compression on balanced trees and takes $O(m\alpha(m, n))$ time, where n is the number of vertices, m is the number of edges in the given graph, and α is a functional inverse of Ackermann's function.

In [13], we proposed an associative parallel algorithm for the same task. Let us explain the main idea of this algorithm.

Let (x_1, y_1), (x_2, y_2), $\dots$, (x_k, y_k) be chords of G with respect to the given MST T. The chords are updated in the increasing order with respect to their weights as follows. First, we define the positions of tree edges which belong to the path (say, γ_1) joining vertices x_1 and y_1. Each edge from γ_1 should be replaced with the chord (x_1, y_1). Let the first r $(r \leq k-1)$ chords have been updated. Then the $(r + 1)$-th chord is updated as follows. We define the positions of tree edges which belong to the path (say, γ_{r+1}) joining the vertices x_{r+1} and y_{r+1}. Each edge from γ_{r+1} not included into the tree paths γ_1, γ_2, $\dots$, γ_r should be replaced with the chord (x_{i+1}, y_{i+1}). On the STAR–machine, this algorithm is implemented as a procedure UpdateMST having the following input parameters: the matrices $Left$, $Right$, $Weight$, and $Code$, the minimum spanning tree given as a slice $Tree$, and the number of vertices n. The procedure returns a matrix F whose every j-th column $(j \leq 2k - 1)$ saves the positions of edges from T replaced with the same chord. Position of this chord is marked with $'1'$ in the $(j + 1)$-th column of the matrix F.

In [1], Chin and Houck proposes the following criterion of verifying minimum spanning trees in undirected graphs:

A spanning tree T is *optimum* if and only if for each chord (v_i, v_j) $wt(v_i, v_j) \geq \max\{wt(x, y)$: (x, y) is on the tree path joining v_i and $v_j\}$.

In [17], this criterion is implemented on the STAR-machine as a procedure CST (Checking a Spanning Tree) having the following input parameters: matrices $Left$, $Right$, $Weight$, and $Code$, a spanning tree T given as a slice $Tree$, and the number of vertices n. The procedure returns **true** if and only if all chords of T satisfy the criterion.

The procedure CST runs as follows. At first, by means of the procedure MatrixPaths, we construct the matrix $TPaths$. Then for every chord (v_i, v_j) by means of the procedure PathPositions, we determine the positions of edges included into the tree path joining the vertices v_i and v_j. Further by means of the basic procedure GREAT, we verify whether there is an edge in this path whose weight is greater than the weight of the chord (v_i, v_j).

In [13], we present an associative parallel algorithm for finding the fundamental set of circuits with respect to the given spanning tree T. On the STAR–machine, this algorithm is given as a procedure FSC (Fundamental Set of Circuits) that uses the same input parameters as a procedure CPT. This procedure returns a matrix F whose every i-th column saves the positions of edges included into the i-th fundamental cycle.

Informally, the procedure FSC runs as follows. At first, by means of a slice, say Z, one saves the positions of chords with respect to a given spanning tree. Then by means of the procedure MatrixPaths, we construct the matrix $TPaths$. While $Z \neq \emptyset$, the current fundamental cycle is determined as follows. The position of the current updated chord (x, y) is selected and deleted from the slice Z. After that by means of the procedure PathPositions, one determines the positions of tree edges that belong to the tree path, say γ, from x to y. The current fundamental cycle is obtained after including the position of the chord γ into the tree path from x to y. These positions are saved in the current bit column of the matrix F.

6. Finding the smallest spanning tree with a degree constraint of a vertex

In this section, we present an improved version of the implementation of the Gabow algorithm for finding the smallest spanning tree with a degree constraint of a vertex on the STAR–machine.

Let t be the degree constraint of a vertex r in a given undirected graph G and $deg(r) = k$. The Gabow algorithm constructs the smallest spanning tree of G that contains only t edges incident to r. Let T_k be the set of all spanning trees for which $deg(r) = k$. Let R be the set of all edges incident to r. The main idea of this algorithm is to find the smallest spanning tree that contains the set R. Then elementary exchanges are performed until the degree of the vertex r decreases to t. To speed up the computation, first of all a minimum spanning forest U of $G - r$ is constructed. Then only edges from $R \bigcup U$ are used. The elementary exchanges satisfy to the following property. Let T be the smallest spanning tree in the set T_k. If edges $e \in T \bigcap R$ and $f \notin T \bigcup R$ are chosen so that $T - e + f$ is a spanning tree and $wt(f) - wt(e)$ is the minimal value among such edges, then $T - e + f$ is the smallest spanning tree in the set T_{k-1}.

To find the elementary exchanges, the Gabow algorithm uses a system of queues. For every edge $e \in T \bigcap R$, a queue $F(e)$ is built to save the edges f that merge two connected components of $T - e$. The priority queue of any edge f is its weight. Another queue X saves elementary changes (e, f), where f is the edge of the minimal weight that can replace the edge e. The priority of this exchange is the value $wt(f) - wt(e)$. Let the edges e and e' from R have the same replacement f. Then after deleting the edge e and replacing it with the edge f, the edge f should be deleted from the queue $F(e')$, the replacement (e', f) should be deleted from the queue X, a new priority queue $F(e')$ should be built by merging the queues $F(e)$ and $F(e')$, a new replacement f' should be found for e' and it should be written into the queue X.

In [8], it is assumed that every edge incident to the vertex r has the form $< r, p, wt(r, p) >$, and all such edges are written in the first r rows of the graph representation. To describe the main constructions, the following three auxiliary procedures are used.

The minimum spanning tree of a connected component is built by means of the procedure MSTC(*Left,Right, Weight,S1,Q,R*) using the slice $S1$ to save the positions of edges from G, among which the connected component is constructed. It returns the slice R to save the positions of edges belonging to the minimum spanning tree of the connected component and the slice Q to save the positions of the edges from $S1$ not included into the minimum spanning tree.

The minimum spanning forest for $G - r$ is built by means of the procedure FOREST$(Left, Right, Weight, Z, U)$, using the slice Z to indicate the positions of edges among which the connected components are looked for. It returns a slice U to save the positions of edges belonging to the minimum spanning forest.

For a given edge $e = (r, p)$, written in the j-th row of the graph representation, the positions of its replacing edges are found by means of the procedure EXCHANGE$(Left, Right, j, Z, R, U)$. It returns a slice U, in which the positions of replacing edges for the edge from the j-th position are marked with bits $'1'$. The slice Z saves the positions of the edges deleted from the minimum spanning forest of $G - r$, and the slice R saves the positions of the edges from the minimum spanning tree containing all the edges incident to

r.

Let us briefly consider four constructions from [4] used in the associative version of the Gabow algorithm.

Construction 7. (*Finding the smallest spanning tree including all edges incident to r.*)

At first, define the maximal weight ($w1$) of the edges incident to r. Then costruct a new matrix $Cost$ from the matrix $Weight$ as follows. Weights of edges, incident to r, are increased by the value $w1$. After that perform the procedure MSTC using the matrix $Cost$ instead of the matrix $Weight$.

This construction uses the basic procedures MAX, ADDC, and TMEDGE.

Construction 8. (*Finding a replacement for an edge from the j-th row.*)

Let an edge e be written in the j-th row of the graph representation. By means of the procedure EXCHANGE, define in a slice (U) the positions of edges that can replace the edge e. Then define the position i of the replacing edge f having the minimal weight. Save the position j of the edge e in a slice (W) and store the position i of its replacing edge f in the j-th component of an array (A).

Construction 8 uses the basic procedures MATCH and MIN.

Construction 9. (*Finding the current exchange.*)

By means of a matrix ($Cost1$), save the augments of rows corresponding to the positions $'1'$ in the slice W. The position of the current exchange (m) is the row of the matrix $Cost1$ with the minimal augment. Thus, the pair $(m, A[m])$ is the current exchange. The position of the replacing edge is obtained by means of the statement `i:=A[m]`.

Construction 9 utilizes the basic procedures SUBTV and MIN.

Construction 10. (*Checking whether there are two edges with the same replacing edge.*)

Let (m, i) be a current exchange. Knowing the position m of the deleted edge e, define the column number (p) in a matrix ($Change$) where the positions of its replacing edges are stored. Write $'0'$ in the i-th bit of the p-th column. Check whether there is bit $'1'$ in the i-th row of the matrix $Change$. If this is true, determine the column number (l) whose i-th bit is $'1'$ in the i-th row of the matrix $Change$ and replace it with $'0'$. Finally, perform the disjunction between the p-th column and the l-th column and write the result into the l-th column of the matrix $Change$.

Now we present a new elegant associative algorithm for performing the procedure EXCHANGE. To this end, we use the matrix $TPaths$ built by means of the procedure MatrixPaths considered in the previous section. From constructing the matrix $TPaths$, we obtain the following property: every i-th row of this matrix saves by bits $'1'$ the positions of the vertices whose tree paths from the root r include the edge written in the i-th row of the graph representation. Hence, if we delete from G the edge $e = (r, p)$ written in the j-th row, then the j-th row of the matrix $TPaths$ saves by $'1'$ the vertices that belong to the subtree with the root p. These vertices form a connected component including p.

The new associative algorithm uses the following idea. Let a slice Z save the positions of edges deleted from the minimum spanning forest for the graph $G - r$. Then the replacement for the edge $e = (r, p)$ can be such an edge marked by $'1'$ in the slice Z that has only a single vertex belonging to the subtree with the root p.

The new associative algorithm uses the matrices $Left$, $Right$, $Code$, and $TPaths$, the position j of the deleted edge e, the slice Z mentioned above and a new slice Q to save the positions of the replacing edges. Initially, the slice Q consists of zeroes and the slice Z saves the positions of edges that have been deleted from the graph G after constructing the minimum spanning forest. The algorithm performs the following steps.

Step 1. Write zeroes into the slice Q. By means of a variable of the type **word**, save the j-th row of the matrix $TPaths$.

Step 2. While $Z \neq \emptyset$, perform the following actions:

- select the position l of the uppermost edge (say, γ) in the slice Z. Then write zero into the l-th bit of this silce;
- determine both end-points of the edge γ written as decimal numbers;
- verify whether only a single end-point of the edge γ belongs to the subtree with the root p. If this is true, then γ is a replacement for the edge e. Therefore write the bit $'1'$ into the l-position of the slice Q.

On the STAR–machine this algorithm is implemented as a procedure EXCHANGE.

```
procedure EXCHANGE(Left,Right: table; Code: table; TPaths: table;
  j: integer; var Z,Q: slice(Left));
var v,v1,v2: word(TPaths);
  Z1: slice(Left);
  X.X1: slice(Code);
  l,k1,k2: integer;
1. Begin CLR(Q); SET(X);
2.  v:=ROW(j,TPaths);
/* The row $v$ saves the numbers of vertices that
  belong to the subtree with the root $p$. */
3.  while SOME(Z) do
4.    begin l:=STEP(Z);
5.      v1:=ROW(l,Left); v2:=ROW(l,Right);
6.      MATCH(Code,X,v1,X1); k1:=FND(X1);
7.      MATCH(Code,X,v2,X1); k2:=FND(X1);
/* The edge $(k1, k2)$ is written in the $l$-th row of
```

```
   the graph representation. */
 8.      if v(k1)='0' and v(k2)='1' then Q(l):='1';
 9.      if v(k1)='1' and v(k2)='0' then Q(l):='1';
10.    end;
11. End.
```

In [8], the associative version of the Gabow algorithm is represented as the main procedure SSTEQ written in the language STAR. Taking into account the improvement of the auxiliary procedure EXCHANGE, it is necessary to change the list of formal parameters of the main procedure SSTEQ and to include the execution of the procedure MatrixPaths immediately before performing the procedure EXCHANGE in the body of the procedure SSTEQ. The modified procedure SSTEQ uses the following parameters: the matrices *Left, Right, Weight, TPaths*, and *Code*, a variable *node* to save the binary code of the selected vertex r, a variable n to save the number of graph vertices, a variable k to save the degree of the vertex r and a variable t to save the degree constraint of r. The procedure returns a slice R that saves the positions of edges belonging to the smallest spanning tree.

In [8], we showed that the procedure SSTEQ takes $O(n \log n)$ time corresponding to the time of performing the procedure of finding the minimum spanning tree on the STAR–machine.

7. Conclusions

We have selected a group of constructions that are used to represent on the STAR–machine the classical graph algorithms of Prim–Dijkstra, Kruskal and Gabow along with the method of finding paths with respect to a given spanning tree. These algorithms take into account the main advantages of vertical processing systems such as access data by contents, data parallelism at the base level, and the use of simple data structures given as 2D tables consisting of binary strings. As shown in [11], associative versions of the Prim–Dijkstra and Kruskal algorithms use only the binary representation of vertices. The new associative version of the Gabow algorithm applies the representation of vertices in binary and decimal codes due to the use of the method for finding tree paths in undirected graphs.

The selected constructions can be used to solve other problems on vertical processing systems.

We are planning to select constructions that are used to represent other classical graph algorithms on the STAR–machine.

References

[1] Chin F., Houck D. Algorithms for updating minimal spanning trees // J. of Computer and System Sciences. – 1978. – Vol. 16. – P. 333–344.

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

[3] Kapralski A. Sequential and Parallel Processing in Depth Search Machines. – Singapore: World Scientific, 1994.

[4] Kokosiński Z. An associative processor for multi–comparand parallel searching and its selected applications // Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications. PDPTA'97, Las Vegas, USA. – 1997. – P. 1434–1442.

[5] Kokosiński Z., Sikora W. An FPGA implementation of multi–comparand multi–search associative processor // Proc. 12th Int. Conf. FPL'2002. – Lect. Notes Comput. Sci. – Springer-Verlag, 2002. – Vol. 2438. – P. 826–835.

[6] Nepomniaschaya A.S., Kokosiński Z. Associative graph processor and its properties // Proc. of the Intern. Conf. PARELEC'2004. – IEEE Computer Society, 2004. – P. 297–302.

[7] Nepomniaschaya A.S. Basic associative parallel algorithms for vertical processing systems // Bull. Novosibirsk Comp. Center. Ser. Comput. Sci. – 2009. – IIS Special Iss. 29. – P. 63–77.

[8] Nepomniaschaya A.S. Efficient associative algorithm for finding the smallest spanning tree of a graph with a degree constraint of a vertex // Cybernetics and System Analysis. – Kiev: Naukova Dumka, 1998. – N 1. – P. 94–103 (In Russian). – (English translation by Plenum Press).

[9] 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.

[10] Nepomniaschaya A.S. An associative version of the Bellman–Ford algorithm for finding the shortest paths in directed graphs // Proc. of the 6-th Intern. Conf. PaCT-2001. – Lect. Notes Comput. Sci. – 2001. – Vol. 2127. – P. 285–292.

[11] Nepomniaschaya A.S. Comparison of performing the Prim–Dijkstra algorithm and the kruskal algorithm by means of associative parallel processors // Cybernetics and System Analysis. – Kiev: Naukova Dumka, 2000. – N 2. – P. 19–27 (In Russian). – (English translation by Plenum Press).

[12] Nepomniaschaya A.S. Associative parallel algorithms for dynamic edge update of minimum spanning trees // Proc. of the 7th Int. Conf. PaCT 2003. – Lect. Notes Comput. Sci. – 2003. – Vol. 2763. – P. 141–150.

[13] Nepomniaschaya A.S. Associative parallel algorithms for computing functions defined on paths in trees // Proc. of the Intern. Conf. on Parallel Computing in Electrical Engineering. – IEEE Computer Society Press, 2002. – P. 399–404.

[14] Nepomniaschaya A.S. Efficient implementation of the Italiano algorithms for updating the transitive closure on associative parallel processors // Fundamenta Informaticae. – IOS Press, 2008. – N 2–3. – P. 313–329.

[15] Nepomniaschaya A.S. Associative version of the Ramalingam decremental algorithm for dynamic updating the single-sink shortest paths subgraph // Proc. of the 10-th Intern. Conf. on Parallel Computing Technologies, PaCT-2009, Novosibirsk, Russia. – Lect. Notes Comput. Sci. – 2009. – Vol. 5698. – P. 257–268.

[16] 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. – N 3. – P. 45–57 (In Russian). – (English translation by Springer).

[17] Nepomniaschaya A.S., Borets T.V. Associative parallel algorithm of checking spanning trees for optimality // Bull. Novosibirsk Comp. Center. Ser. Computer Science. – Novosibirsk, 2002. – Iss. 17. – P. 75–88.

[18] Parhami B. Search and data selection algorithms for associative processors // Associative Processing and Processors. – IEEE Computer Society, 1997. – P. 10–25.

[19] Potter J.L. Associative computing: a programming paradigm for massively parallel computers. – New York and London: Plenum Press, 1992.

[20] Tarjan R.E. Applications of path compression on balanced trees // J. of the ACM. – 1979. – Vol. 26, N 4. – P. 690–715.