Asynchronous Distributed Algorithms for Static and Dynamic Directed Rooted Graphs

[1]I.B. Burdonov < igor@ispras.ru >
[1]A.S. Kossatchev <kos@ispras.ru>
[1,2,4]V.V. Kuliamin < kuliamin@ispras.ru>
[1,2]A.N. Tomilin <tom@ispras.ru>
[1,3]V.Z. Shnitman <vzs@ispras.ru>
[1]Ivannikov Institute for System Programming of the RAS,
25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia
[2]Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia
[3]Moscow Institute of Physics and Technology (State University),
9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia
[4]National Research University Higher School of Economics (HSE)
11 Myasnitskaya Ulitsa, Moscow, 101000, Russia

Abstract. The paper provides a review of distributed graph algorithms research conducted by authors. We consider an asynchronous distributed system model represented by a strongly connected directed rooted graph with bounded edge capacity (in a sense that only a bounded number of messages can be sent through an edge in a given time interval). A graph can be static or dynamic, i.e. changing. For a static graph we propose a spanning (in- and out-) tree construction algorithm of time complexity $\boldsymbol{O}(n / k + d)$, requiring $\boldsymbol{O}(n\, d \log \Delta^+)$ message size and the same size of memory of each computing agent located in graph vertex, where n is the number of vertices of the graph, k is the capacity of an edge, d is the maximum length of simple path in the graph, Δ^+ is the maximum outdegree of the vertices. The spanning trees constructed can be used in distributed computation of a function of the multiset of values assigned to graph vertices in a time not greater than $3d$. In a dynamic graph we suppose that $k = 1$ and an edge can appear, disappear, or change its end. We propose a dynamic graph monitoring algorithm than delivers information on any change to the root of the graph in $\boldsymbol{O}(n)$ or $\boldsymbol{O}(d)$ after the changes are stopped. We also propose graph exploration and marking algorithm with time complexity $\boldsymbol{O}(n)$. The marking provided by it is used in distributed computation of a function of the multiset of values assigned to dynamic graph vertices, which can be performed in time $\boldsymbol{O}(n^2)$ with messages of size $\boldsymbol{O}(\log n)$ or in time $\boldsymbol{O}(n)$ with messages of size $\boldsymbol{O}(n \log n)$.

Keywords: distributed algorithms; asynchronous systems; directed graph; rooted graph; dynamic graph; parallel computations

DOI: 10.15514/ISPRAS-2018-30(1)-5

For citation: Burdonov I.B., Kossatchev A.S., Kuliamin V.V., Tomilin A.N., Shnitman V.Z Asynchronous Distributed Algorithms for Static and Dynamic Directed Rooted Graphs. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 1, 2018, pp. 69-88. DOI: 10.15514/ISPRAS-2018-30(1)-5

1. Introduction

This paper considers distributed graph algorithms. To treat them formally, we must use a formal computation model, and the choice of such a model critically influences the complexity of possible algorithms. In this work we use the following model: a set of computing agents located in the vertices of the graph under consideration (an algorithm is intended to compute some information about this graph) and interacting by passing messages through graph arcs [1–3].

Such computing agents are called in various ways in the literature: tasks [1], processors [2], processes [3], or state machines [4] (if they are actually state machines, maybe not finite ones). We suppose that the agents can interact only in a peer-to-peer way. A round of agent's operation can include message reception, changing the agent's state, and sending another message. In [1] such an agent is called reactive entity or message-driven entity. To be able to send a message through some arc, an agent should refer to this arc. To make this possible, all arcs incident to one vertex are supposed to be enumerated, so an agent just use the arc's number. In case of directed graph, this enumeration includes only arcs outgoing from the vertex. We also suppose that am agent has no information on other vertices – it knows only the number of outgoing arcs in its vertex, and the vertex identifier if the graph is enumerated.

Here and in [1] computations on directed graph are considered, although computations on undirected graph are more usual, see [2, 3]. Note, that algorithms solving the same tasks on directed graphs are considerably more complex. An example is provided by the task of graph traversal by a single message, with agents in the vertices being finite state machines. For undirected graphs it is solved by well-known Tarry algorithm [2, 22] with time complexity O(m), where m is the number of graph arcs. This complexity is minimal, because it coincides with lower time bound for any algorithm for this task. For directed graphs lower time bound for the same task is O(nm), where n is the number of graph vertices, and the best known algorithm has time complexity O(nm + n2 log log n) [16]. It is still unknown whether the gap between lower bound estimate and actual algorithm complexity can be reduced for directed graphs.

Synchrony or asynchrony of computation agents is also an important aspect of the model. In synchronous models the agents work in lockstep mode, i.e., all the agents make one computation step simultaneously, so the main complexity is related with number of communication acts [18, 19]. In asynchronous model, the time of message transport can differ for different arcs and different messages. Usually the time of message transport through one arc is assumed to be bounded by some

constant, which can be taken as 1 tic. The algorithm time complexity in asynchronous model is the worst case time of its operation. Also in asynchronous model, an arc becomes a buffer of messages sent by one arc end and not yet accepted by the other, usually organized as a queue. This queue can be unbounded or can has some maximum number of messages that can be stored, called an arc capacity. It is so an important characteristic of a model [1]. In this work, we consider a model with bounded arc capacity.

We also assume that graphs can have multiple arcs leading from one vertex to another one, and can have loops leading from a vertex to itself (some authors called such graphs pseudographs or multigraphs). In opposite to our approach, most works on distributed computations consider graphs that cannot have multiple arcs or loops.

For distributed asynchronous algorithms, the memory used is an important characteristic. It is true both for the inner memory of a computation agent (which can be regarded as a logarithm of the number of its states) and for the size of messages used in communication. Therefore, further we estimate the inner memory of agents and the size of messages used in an algorithm as functions of the graph complexity.

In synchronous models all the agents start their operation simultaneously and the end of operation is often not specified explicitly. In asynchronous models the algorithm operation is initiated by an external Start message, which is accepted by an agent in some graph vertex, which than starts sending messages to other vertices, agents in which start their operation after accepting them [3]. We assume that a starting vertex is fixed, it is called the root of a graph.

The other problem with distributed computation is termination detection [2] – how to notice the end of algorithm operation. We assume that the algorithm is finished when the root agent sends to the environment special Finish message, possibly containing some result data. A procedure helping the root agent to learn that the algorithm work is finished is nontrivial in asynchronous models.

One of important tasks of distributed computations is computation of a global function on a graph [3]. In this work, we consider a task of computing a function of a multiset of values assigned to graph vertices. An agent located in a vertex knows the value assigned to it. To compute the global function value we use the notion of aggregate function, which global value can be computed from values for nonintersecting subsets of vertices, and minimal aggregate extension, which exists for any function [11, 12, 14, 21].

Such computation tasks are usually solved with the help of broadcast and convergecast operations. The broadcast problem is a problem of communication organization from one vertex to all other vertices. The convergecast problem is a problem of communication organization from all vertices to some single vertex. To solve broadcast and convergecast problems a spanning tree of the graph is used. In a directed graph broadcast is performed as query message passing along forward or spanning out-tree, which has the root coinciding with the root of the graph and arcs

leading from an ancestor vertex to its descendants. Convergecast is performed as answer message passing along backward or spanning in-tree, with arcs leading from descendant vertices to their ancestor, and finally to the root. To use spanning in-tree for correct convergecast an agent in a vertex should know the number of incoming backward arcs, because it can send its answer only after accepting all the answers of agents located in the descendants.

Algorithm of spanning tree construction for undirected graph uses information propagation with feedback [3]. It is performed by broadcast with the help of flooding (sending messages along all arcs that are not used before), and then convergecast is performed to mark out the spanning tree arcs and assign the number of descendants to each vertex. This algorithm uses the possibility to send a message along an arc, which is used to transfer a previously accepted message. Directed graph has no such a possibility. Instead, one can use for the same task a path from the arc's end to the root and a path from the root to the arc's start. To construct these paths, broadcast messages should accumulate the description of the path passed as a sequence of arc numbers. In addition, convergecast also uses flooding, but takes into account that several floodings initiated by different vertices are working in parallel on the graph.

In many applications of distributed computations on graphs, including communication networks, VLSI design, assembly planning, the base graph can be slightly changed from time to time, that is, its vertices or arcs can be inserted or deleted. In the last decade, there is a growing interest to algorithms solving certain tasks on such dynamic graphs. If one allows only arc insertion or deletion, keeping the set of vertices stable, such a graph is described by dynamic connectivity structure, a data structure that dynamically maintains information about the connected vertices. Some works consider a case when the set of vertices can be also changed [20].

The following problems are often considered on dynamic graphs: connectivity determination, computation of the shortest path between the given vertices, spanning tree construction, etc. The usual goal of dynamic graph algorithms is to update efficiently the solution of a problem after changes, rather than having to re-compute it from scratch each time. Such algorithms provide incremental updates for the previous solution. Our approach is different; we consider possibility to solve some task without any previously gathered data on a graph, which is still changing. We treat directed dynamic graphs with stable vertices set and changing arcs, an arc can disappear, appear, or change its end. We also consider a problem of graph monitoring: how to gather actual information on graph structure in its root agent. Of course, this information gathering have sense if changes are not too quick. We give an estimate of time needed for graph structure information update in the root agent, so if no changes occur on this interval, one can trust the data gathered.

Another problem on dynamic graphs considered here is computation of a function on a multiset of values assigned to graph vertices. In this case, we cannot use

spanning trees because of possible changes in graph structure. In such a setting one can send queries to all vertices from the root and then gather the answers, but answers from different vertices should be distinguished. There are different ways to distinguish them. One technique uses static enumeration of vertices and their identifiers, in this case computation complexity is O(n) and message size is O(n log n). Another technique uses linear order on vertices and sending answers according to this order, in this situation message size can be O(log n), but total computation time becomes O(n2). These methods can be generalized, namely, one can construct virtual spanning tree of the graph with the help of additional virtual arcs and use it to gather answers from vertices. If such a spanning tree has w leaf vertices, then messages used can be of size O(w log n) and total computation time is O(n2 / w).

Below in Section 2 we consider static graphs, their marking out algorithm, and parallel computation on static graph. Section 3 presents the dynamic graphs, monitoring algorithm, and parallel computation on dynamic graph. The conclusion summarizes the paper.

2. Static Graphs

In this section we suppose that distributed computation is performed by agents located in vertices of a directed graph and interacting by passing messages through graph arcs, along their direction. The graph is static, i.e., stays unchanged during the computation. An agent has no information on graph structure, it knows only the number of arcs outgoing from its vertex. Outgoing arcs are numbered, so an agent can send a message along an arc using its number.

Agents operation is asynchronous. On single step of its operation, an agent can accept all messages sent to it through incoming arcs, perform some internal computations, and send several messages through some outgoing arcs. An arc is a buffer with capacity k, i.e., in one time not more than k messages can be sent along this arc and not yet accepted. If an arc already has k unaccepted messages, an agent located in its starting vertex cannot send new messages along it. We suppose that message size is bounded, so arc capacity put a bound on the size of stored data. This can be implemented in two ways: one can has k messages of some constant size on an arc or one message of a size that is k time bigger. The difference is that in the first case it is possible to send more messages, if the number of unaccepted messages on an arc is less than k. We consider here algorithms using the first option, they can send several messages of bounded size along one arc, but not greater than k in a time.

A message can be sent along only one arc, along all outgoing arcs, or along some subset of outgoing arcs. Nevertheless, we assume that a message waits until all outgoing arcs are capable to transport it. This assumption can increase the total computation time, but make unnecessary separate signals about ability of arcs to

carry a message, a single signal concerning all arcs (that a message can be sent along any of them) is sufficient.

To estimate time complexity of computation we count time of internal operation of an agent negligible. We also assume that a message is transported through an arc in one tic, i.e., not later than one tic after sending it is accepted on the end of the arc.

We also use the following notation: n is the number of graph vertices, m is the number of graph arcs, d is the maximum length of a simple path (passing each vertex not more than once), $\Delta+$ is the maximum number of arcs outgoing from one vertex, $\Delta-$ is the maximum number of arcs incoming to one vertex.

2.1 Marking out a graph

We consider in this subsection a problem of graph exploration. The solution of this problem should be an algorithm that starts by accepting an external Start message in the root agent, performs marking out of the graph by computing and storing in agents' internal memory some data that provides a local information on graph structure, and finishes by sending Finish message from the root agent as an answer to the Start one.

The graph marking out includes the following.

Spanning out-tree, directed from the root. We call its arcs forward arcs. An agent located in each vertex should store all the outgoing forward arcs. Other outgoing arcs are called chords.

Spanning in-tree, directed to the root. Its arcs are called backward arcs. A vertex can have not greater than one outgoing backward arc (the root has no one). An agent located in the vertex (except for the root) should store the number (id) of the backward arc.

An agent located in the vertex should store the number of backward arcs incoming to it. This number is necessary to gather correctly information along the in-tree, the messages accepted along backward arcs are counted until the counter reaches this number that means that all the data from the preceding part of in-tree are already collected.

This marking out can be considered as a result of graph exploration, sufficient to perform further exploration of the graph or distributed computations on it in an efficient way.

The algorithm of marking out is described in details and with proofs in [10, 11]. Here we provide only the general description of it and give complexity estimates without proofs. The algorithm has three phases: construction of spanning out-and in-trees, construction termination detection, and setting counters of incoming backward arcs.

2.1.1 Construction of spanning out-and in-trees

This step uses 4 message types: Start is used to initiate the operation in the root and to set vertex identifies, Search is used to find paths from any vertex to the root, Forward marks out the spanning out-tree, Backward marks out the spanning in-tree.

The root agent accepts Start message from the environment and this initiates the graph marking out. Start message is sent further with flooding pattern: an agent accepts the first Start message, sends it along all outgoing arcs, and ignores all other incoming Start messages (accepts them and does nothing in addition). A message of this kind accumulates a vector – the sequence of arc numbers used along its way (each agent before sending the message further appends the sequence in it with the number of the arc it uses to send it). The vector of the first Start message accepted in the vertex becomes the identifier of this vertex. The root identifier is an empty vector.

A Search message looks for a path from a vertex to the root. An agent that accepts the first Start message creates and sends Search messages along all outgoing arcs. Such a message contains an initiator vector (the identifier of the vertex where it is created) and accumulates backward vector – the sequence of arc numbers it passes. An agent stores internally the set of identifiers of initiators of Search messages it already processed. After accepting Search message, a non-root agent looks for its initiator identifier in this set. If the initiator is already processed, the message is ignored. Else, the initiator identifier is stored and the agent sends Search messages along all outgoing arcs with their backward vectors appended by numbers of corresponding arcs.

The root agent accepting Search message creates Forward message putting the initiator vector and backward vector in it. Forward message is moved along the path described by the initiator vector. To make this possible, each agent before sending the message along the first arc of the vector marks this arc as a forward one and removes its number from the vector in the message. If an agent accepts the Forward message with empty initiator vector, then it is the initiator. It creates Backward message, put backward vector from the Search message in it, and sends it along backward path.

Backward message moves along backward path in the same way – each agent accepting it before sending it along the first arc of the backward vector, marks this arc as a backward one (stores its number in a special memory slot), and removes its number from the backward vector in the message. Backward arc number can be already stored in a vertex agent; in this case, the agent rewrites it. This guarantees that no cycles along backward arcs appears in the end.

2.1.2 Construction termination detection

Construction termination detection is based on counting of arc starts and ends, when this numbers become equal all the arcs are explored and so the construction step is complete. To implement this idea the root agent stores the value of known arc starts

minus known arc ends. After the operation start, the counter equals to the number of outgoing arcs from the root.

To count arc starts every Search message has an additional field storing the number of arcs outgoing from the message initiator. After accepting such a message, the root agent adds this field value to the counter.

To count arc ends we add two special message types, Finish and Minus. All agents have the counter of incoming arcs. When Forward message is accepted by its goal agent (the initiator), it sends Finish messages along all outgoing arcs. The root agent does this just after sending Start messages. When Finish message is accepted, the agent increments the counter of incoming arcs. Minus message is created and sent to the root agent along the backward arc just after the backward arcs is set in the vertex for the first time. The first Minus message contains the value of the counter of incoming arcs. If a Finish message is accepted after sending the first Minus message, additional Minus message with value 1 is created and sent to the root agent. When the root agent accepts Minus message, it decreases its counter of arc starts on the value of the message field. When this counter becomes 0, the construction step is finished.

2.1.3 Setting counters of incoming backward arcs

After the construction of in-tree is finished, it becomes possible to set the counters of incoming backward arcs. For this purpose two additional message kinds, Begin and End are used. Begin messages are created by the root agent and broadcasted along the out-tree – each agent after accepting such a message creates its copies and send them along all outgoing forward arcs. It also creates and sends along the backward arc End message with initial flag set to true.

After accepting End message with raised initial flag, an agent increments its incoming backward arcs counter, sets the flag to false, and sends the message along backward arc (if it isn't the root agent). End messages with dropped initial flag are just sent along backward arcs. The root agent counts accepted End messages. When their number reaches the number of registered initiators, the counter setting and so the whole algorithm operation are finished.

2.1.4 Algorithm complexity

The time complexity of the algorithm is $O(n / k + d)$. The size of internal agent memory used is $O(nd \log \Delta+)$. The maximum size of message used is $O(d \log \Delta+)$. Note, that here d means the maximum length of non-intersecting path, not the graph diameter. This is a consequence of asynchrony; the forward path from the root to some vertex may appear to be the maximal non-intersecting path, not a path of minimum possible length between these two vertices.

2.2 Parallel computations and aggregate functions

In this subsection we consider the problem of parallel computation of a value of a function on graph. We suppose that each graph vertex has some value assigned to it (vertex agent has special operation providing this value) and we need to compute the value of some function on this multiset of values.

Let X denote a set, from which values assigned to graph vertices are taken. X# is a set of all finite multisets of elements from X. Multisets are considered, because different graph vertices may have equal values assigned.

A function $g : X\# \to Y$ is called aggregate when $\exists e : Y \to Y \ \forall a, b \in X\#$ $g(a \cup b) = e(g(a), g(b))$. That is, g value on a multiset can be computed by parts. In this situation e is called an aggregator of g.

Examples of aggregate functions are the following: sum $\Sigma : \{a1,...,an\} \to a1 +...+ an$ has $e : (a, b) \to a + b$, mimimum $\min : \{a1,...,an\} \to \min(a1,...,an)$ has $e : (a, b) \to \min(a, b)$, sum of squares $Q : \{a1,...,an\} \to a12 +...+an2$ has $e : (a, b) \to a + b$.

Not all the functions are aggregate, for example, arithmetical mean is not an aggregate function. But one can expand function $f : X\# \to Y$ as a composition $f = hg$, where $g : X\# \to Z$ is aggregate and $h : Z \to Y$ is just some function. In this case g is called an aggregate extension of f. An aggregate extension can help to compute f by parts, by computing g by parts and then computing h once in the end. Some extensions aren't helpful actually, e.g., one can take an identity function on X# as g and f itself as h. To prevent this situation, we use minimal aggregate extension. Intuitively, minimal aggregate extension keeps minimum information to make possible further computation of f. Formally, $g : X\# \to B$ is a minimal aggregate extension of $f : X\# \to Y$, if g is its aggregate extension and for each g': $X\# \to B'$, which is an aggregate extension of f there is $j : B' \to B$, such that $g = jg'$. Minimal aggregate extension exists for every function on multisets and is unique up to isomorphism.

Examples of minimal aggregate extension are the following: for arithmetical mean function $f : \{a1,...,an\} \to (a1 + ... + an) / n$ the minimal aggregate extension is provided by function $g : \{a1,...,an\} \to ((a1 + ... + an), n)$ and corresponding $h : (a, n) \to a / n$, for root mean square $f : \{a1,...,an\} \to ((a12 +...+an2) / n)1/2$ the minimal aggregate extension is provided by function $h : (a, n) \to (a / n)1/2$.

Aggregate functions theory is a modification of theory of inductive functions on finite sequences, see [21]. Detailed proofs can be found in [11].

2.3 Parallel computations on static graph

In this subsection we describe an algorithm for computation of a function of values assigned to graph vertices (details can be found in'[11, 12]). This algorithm uses the marking out provided by the algorithm described above: spanning in-and out-trees

and counters of incoming backward arcs. Note, that this marking out can be constructed once and further used for many computations.

Computation is started when the root agent accepts an external **Start** message with specification of three functions *h*, *g*, *e*. One need to compute the value of $f = hg$ on the multiset of values from *X* assigned to graph vertices. *g* is a minimal aggregate extension of *f*, and *e* is an aggregator function of *g*. Specification of a function can be actually a program to compute it.

The computation is based on broadcast along spanning out-tree and convergecast along spanning in-tree of the graph. **Request** messages containing specifications of *g* and *e* are transferred along forward arcs. **Response** messages are transferred along backward arcs and contain value of *g* on a subset of values assigned to vertices of a subtree of in-tree.

When an agent of a leaf vertex of in-tree (it has incoming backward arcs counter equal to zero) accepts **Request**, it computes the function *g* of the value assigned to its vertex, and sends the computed value in the field of **Response** message along the backward arc. Agent of a non-leaf vertex also computes the value of *g*, but doesn't send it until getting **Response** messages from all incoming backward arcs, using the counter to detect this situation. When it gets all the responses, it can compute the complete value of *g* for the subtree having the corresponding vertex as its root, and then sends the result with **Response** message along its backward arc. When the root agent get all the responses, it can compute the value of *f* by applying *h* to the value of *g*, and sends the final value to the environment.

The total computation time of this algorithm is $\boldsymbol{O}(3d)$, ignoring the computation complexity of *g*, *h*, *e*, the memory size of an agent is bounded by $\boldsymbol{O}(\Delta^+ + log\ \Delta^- + x + y)$, message size is $\boldsymbol{O}(x + y)$, where *x* is the size of specification of *g* and *e*, and *y* is the size of *g* values.

3. Dynamic graphs

In this section we consider computations on dynamic graphs, i.e., graphs, which arcs can change while the set of vertices doesn't change. There are three possible changes of arcs.

- An arc can appear. This event produces a special signal to arc start agent; the appearance signal contains the number of new arc.
- An arc can disappear. We assume that if there is a message being transported along the arc, the arc start agent gets a signal with the number of disappeared arc, and all the messages sent along it and not yet accepted also disappear. If there are no such messages, then no signal is generated. However, in case the agent tries to send a message along the disappeared arc, it also gets a disappearance signal.
- An arc can change its end vertex. No signal is generated on this event.

This model requires minimal number of additional signals to provide agents with data on arc changes. We assume that a special release signal is sent to an agent in the arcs starting vertex when a message of the arc is accepted and new message can be sent along it. We also suppose that arc capacity is 1 – only single message can be transported at a time.

If an arc changes are too frequent, two unprocessed signals can emerge, and in this case we assume that one of them is lost. The second signal can be only an appearance signal, so the rules of signal collapse are as follows.

– From two appearance signals only the last one is retained.
– From disappearance and appearance signals only the appearance one is retained.
– From release and appearance signals any one can be retained.

We ignore the time of agent's internal operation and consider the time of one message transport through an arc as bounded by one tic. Too frequent changes can make impossible message transport along an arc; to prevent this we suppose that some arcs are *long-living*. A long-living arc always transfers at least one message while it is in a stable state. So, long-living arcs are changed not more frequent than once in a tic. We also suppose that changes preserve strong connectedness of the graph. More precisely, the subgraph made of long-living arcs is always strongly connected and contains all the vertices of the graph.

The algorithms presented below are operating when graph is changing, so they use the following heuristics. All the data an agent need to send another agent should be put in a single message, because only a single message is guaranteed to be transported along an arc. Also, a message should be sent each time it becomes possible, that is, the necessary arc appears or is released. Otherwise, the message data may not reach other agents because the arc may change its end several times without any signals to the sender.

3.1 Dynamic graph monitoring

We assume that the graph is enumerated, its vertices has numbers from 1 to n, and each agent knows the corresponding vertex number.

We consider the problem of graph monitoring: to collect complete information on graph structure in the root agent (this information is also gathered in all other vertex agents). Due to permanent changes, one cannot guarantee the correctness of this information, but we can require that the information on arc change becomes known to all agents not later than in some finite time T_0. So, if in time T_0 an arc doesn't change, the agents has correct information about it.

The algorithm is described in details and with proofs in [13], here we provide only the general description and complexity estimates without proofs.

3.1.1 Arc description and messages

The algorithm works as follows. Each agent has an internal storage of arc descriptions. An arc description is a triple of the number of start vertex, the arc number in its start, and the number of the end vertex (if it is known, else it is replaced with 0). Each message also contains a set of arc descriptions. Each time an agent accepts a message, it compares his internal arc descriptions with the ones in the message, copies to its internal memory the descriptions of arcs it doesn't know, and replaces the descriptions, which are more recent in the message. After that, it sends its internal set of descriptions in messages along all outgoing arcs.

In case an arc appears, the start vertex agent creates the new arc description with zero end number. The start vertex agent is also the first one who learns about arc disappearance. In case of arc end change the end agent can discover this, if it accepts a message with another end number (or zero) in arc description. To make this noticeable, the message contains the identifier of the arc used to transfer it.

3.1.2 Arc rank

Since an arc can change several times, an agent should determine whether it has more recent arc description than the one in a message or not. For this purpose, we use a number field in arc description called *rank* and showing "a version number" of the description. When the rank is greater, the corresponding description is more recent and should replace an older one. When an agent learns about a new arc or arc change and creates or modifies an arc description, it should increase the rank in it.

To be sure that in every other description the rank is smaller, it isn't always enough to increase it in a new description by 1. Several arc end changes can lead to the situation when different end agents assign to a new description rank value greater by 1 than the old one. To resolve this, the arc start agent accepting a message with newer arc description always adds additional 1 to updated arc description in its storage. Accordingly, accepting an appearance or disappearance signal the arc start agent increases the arc description rank by 2.

3.1.3 Estimates

The estimate of time needed to an agent to learn about arc change is $T_0 = \boldsymbol{O}(n)$. The estimate of time needed to transport information on all changes after they stop is $T_1 = \boldsymbol{O}(d)$. The size of agent's internal memory and the message size are both $\boldsymbol{O}(m \log \Delta^+ nv)$, where v is the maximum number of changes of one arc. This is caused by the fact the each agent actually stores the full description of graph structure and the same full description is contained in a message. The multiplier v is caused by the arc rank field in messages.

In case when all arc are long-living, we can eliminate v multiplier. In such a situation, the number of arc changes is not greater than total time of operation, which is $\boldsymbol{O}(n)$. So, one can count ranks modulo some number of the value $\boldsymbol{O}(n)$, and the estimate of message size becomes $\boldsymbol{O}(m \log \Delta^+ n)$.

3.2 Parallel computations on a dynamic graph

For parallel computation on dynamic graph, we need to mark it out, which is not performed by monitoring algorithm. Moreover, one cannot use the same method with broadcast of **Request** and convergecast of **Response** messages, because changes in arcs of in-or out-trees may require modifications in spanning trees. Also, successful message transport along arcs is not guaranteed.

However, as one can see on monitoring algorithm example, assumption on arcs long-liveness can guarantee that information from any vertex can be delivered to any other vertex. To do this, vertex agents should send all gathered information along any arc any time the arc is able to transfer it.

This approach is sufficient to deliver **Request** messages to all vertices. One can gather responses in the same way in the root and perform all the computations by the root agent, but this requires very large messages that should contain all the data from all known vertices. Use of spanning in-tree and backward arc counters helps to minimize the size of **Response** messages; along backward arcs one can send only the summary information on the subtree, having the start vertex as its root.

In case of dynamic graph the idea is to use a virtual spanning in-tree, constructed using additional virtual arcs connecting arbitrary vertices. In addition, we can choose the form of the virtual in-tree, since its height h (the maximum length of a path from its leaf to the root) determines the computation time (the computations can be performed in parallel on different paths only, on one path to the root they are performed sequentially) and its width w (the number of leafs) determines the maximum message size.

For the given n and w one can construct the tree of minimum height with n vertices and w leafs. It is a fan-like tree homeomorphic to star graph with w rays (branches), each ray contains h or $h - 1$ vertices, where $h = \lceil (n - 1) / w \rceil$. The vertices on such a tree (besides the root) can be enumerated by two-component index, the ray number from 1 to w and the number of the vertex on the ray from 1 to h, smaller numbers are closer to the root. The backward arc counter in such a tree is necessary only in the root (and it has value w). Other agents should know only the status of its vertex whether it is a leaf (counter value is 0) or not (counter value is 1).

Below we present a general description of the algorithm and its complexity estimates. Details and proofs can be found in [14].

3.2.1 Marking out a dynamic graph

Marking is performed in two phases; at the first phase the root agent gathers data on all vertices using **Forward** messages, at the second one the root constructs the virtual spanning in-tree and sends it to all the vertices using **Backward** messages.

The root agent after accepting the **Start** message from the environment creates **Forward** message, which further circulates on the graph accumulating the arc descriptions (with the same fields as in monitoring algorithm). Any agent accepting this message updates its internal storage with its data, updates the message with the

descriptions, which are more recent in its storage, and sends the updated message along all outgoing arcs.

We need additional assumptions to guarantee that the root agent learns about all the vertices. First, we assume that in the clash of release and appearance signals the release signal is retained. This helps to guarantee the successful transport of a message, since release signal cannot be lost now. Second, we need more stable *initial arcs*. Initial arc is stable from the start of algorithm work until the first message is transported along it. We also assume that all the vertices are reachable from the root along initial arcs.

Using this assumptions an agent can register only arcs appearing until it accepts the first message (and ignore arc appearing signals after this). This helps to register all the initial arcs, and hence all the vertices becomes known to the root agent after some time. The root agent can check the condition that it knows ends of all known arcs, and due to assumptions, this condition is sufficient to conclude that it knows all vertices.

When the root agent learns about all vertices, the second phase starts. The root agent creates the description of virtual fan-like tree, which is sent in **Backward** message. The tree description consists of descriptions of vertices. Each vertex description includes the vertex number, its index in the tree, and the flag stating whether it is a leaf vertex or not. Each agent accepting **Backward** message first time, stores the description of its vertex, removes it from the tree description, stores the modified description, and sends it with **Backward** message along all outgoing arcs. It also stops resending of **Forward** messages. Accepting **Backward** message second time or further, it constructs and stores the intersection of sets of vertex descriptions stored internally and in the message, and stores and sends this intersection. After some time the description of virtual tree in **Backward** messages becomes empty, and then this description in the root agent becomes empty. This is the end of the marking of the graph.

3.2.2 Function computation

To compute a function of a multiset of values assigned to graph vertices we use request-response scheme, as for static graph. But in case of dynamic graph, all the information should be sent in one message along all the arcs that can transport it (that is, each time just after appearance or release signals). Therefore, each request message should also contain a partial response. One response value for each tree branch can be contained in a message, and it is accompanied by the index of the agent, which computed this response. The response value is at first set by the agents located in the leafs of virtual trees, then it is modified by the agents of the next virtual tree vertices in the direction to the root, until the root agent gathers responses from all the root neighbors in the virtual tree. Accepting **Request** message, an agent also stops to send **Forward** and **Backward** messages.

After the end of computation, **Request** messages can still move along the arcs. Their movement can be stopped by next computation, to do this the root agent can enumerate computation tasks requested by the environment and put the number of task in **Request** message. So, **Request** message contains the number of task, the specification of g and e functions, the set of pairs (the value of g for i-th branch, the index of response creator).

3.2.3 Estimates

The time of graph marking out is $\boldsymbol{O}(n)$, the time of function computation is $\boldsymbol{O}(n^2 / w)$. The size of agent internal memory and the size of a message is $\boldsymbol{O}(m \log n\Delta^+)$ on the marking out phase and $\boldsymbol{O}(x + \log N + wy + w \log n)$, where N is the maximum number of tasks, x is the size of specification of g and e, and y is the size of g values. For $w = 1$ one have total computation time $\boldsymbol{O}(n^2)$ and message size $\boldsymbol{O}(x + \log N + y + \log n)$, for $w = n$ one have minimal total computation time $\boldsymbol{O}(n)$ and maximal message size $\boldsymbol{O}(x + \log N + ny + n \log n)$. To remove dependence on N, one can use the same modification as in monitoring algorithm if all arcs are long-living, task number can be countered modulo some number of value $\boldsymbol{O}(n^2 / w)$.

4. Conclusion

The paper presents algorithms for distributed computation of functions on directed graphs, which is performed by agents assigned to graph vertices, communicating by message passing along directed arcs, and knowing only the number of arcs outgoing from the corresponding vertex.

At first we consider static graph, and describe an algorithm that performs the marking out of the graph preparing it for further computations. The marking includes marks of forward arcs, making up a spanning out-tree, along which all the vertices can be reached from the root, marks of backward arcs, making up a spanning in-tree, along which responses from all the vertices can be gathered in the root, and incoming backward arcs counters, which are used to collect responses correctly. The time complexity of the marking algorithm is $\boldsymbol{O}(n / k + d)$. The size of internal agent memory and message size used are $\boldsymbol{O}(nd \log \Delta^+)$. The computation algorithm for a function of multiset of values assigned to graph vertices uses the marking constructed to complete computations in $\boldsymbol{O}(d)$. Here n is the number of vertices, d is the length of maximal non-self-intersecting path, Δ^+ is maximum outdegree, k is the arc capacity, the maximum number of messages that can be sent along an arc and not yet accepted.

Next we concern parallel computations on a dynamic graph that can change its structure during computation. We present a graph monitoring algorithm providing information about most recent changes to all the vertex agents in $\boldsymbol{O}(n)$. We also describe a parameterized algorithm for parallel computation of a function on a dynamic graph. The algorithm parameter w can be chosen from the interval from 1 to n and helps to balance computation time and the size of messages used. The time

of computation is $\boldsymbol{O}(n^2 / w)$ and the message size and agent memory size used is $\boldsymbol{O}(w \log n)$ (ignoring memory for function specification and result storage).

References

[1]. Barbosa V.C. An introduction to distributed algorithms. MIT Press, Cambridge, MA, USA, 1996

[2]. Kshemkalyani A.D., Singhal M. Distributed Computing: Principles, Algorithms, and Systems. Cambridge University Press, March 2011

[3]. Raynal M. Distributed Algorithms for Message-Passing Systems. Springer Publishing Company, Incorporated, 2013

[4]. Schneider F.B. The State Machine Approach. A Tutorial. Fault-Tolerant Distributed Computing. LNCS 448, 1990, pp. 18-41

[5]. Burdonov I.B., Kossatchev A.S. Testing of automata system. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 1, 2016, pp. 103-130. DOI: 10.15514/ISPRAS2016-28(1)-7 (in Russian).

[6]. Burdonov I.B., Kossatchev A.S. Automata system: composition according to graph of links. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 1, 2016, pp. 131-150. DOI: 10.15514/ISPRAS-2016-28(1)-8 (in Russian).

[7]. Burdonov I.B., Kossatchev A.S. Automata system: determinism conditions and testing. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 1, 2016, pp. 151-184. DOI: 10.15514/ISPRAS-2016-28(1)-9 (in Russian).

[8]. Burdonov I.B., Kossatchev A.S. Testing of automata system. Vestnik Tomskogo gosudarstvennogo universiteta. [Bulletin of Tomsk State University. Management, Computer Science and Informatics], № 1, 2017, pp. 67-75 (in Russian).

[9]. Burdonov I.B., Kossatchev A.S. Generalized model of automata system. Vestnik Tomskogo gosudarstvennogo universiteta. [Bulletin of Tomsk State University. Management, Computer Science and Informatics], № 4(37), 2016, pp. 89-97 (in Russian).

[10]. Burdonov I.B., Kossatchev A.S. Buidling direct and back spanning trees by automata on a graph. Trudy ISP RAN/Proc. ISP RAS, vol. 26, issue 6, 2014, pp. 57-62. DOI: 10.15514/ISPRAS-2014-26(6)-4

[11]. Burdonov I.B., Kossatchev A.S., Kuliamin V.V. Parallel computations on a graph. Programming and Computer Software, vol. 41, № 1, 2015, pp. 1-13. DOI: 10.1134/S0361768815010028.

[12]. Burdonov I.B., Kossatchev A.S., Kuliamin V.V. Parallel calculations by automata on direct and back spanning trees of a graph. Trudy ISP RAN/Proc. ISP RAS, vol. 26, issue 6, 2014, pp 63-66. DOI: 10.15514/ISPRAS-2014-26(6)-5

[13]. Burdonov I.B., Kossatchev A.S. Monitoring of dynamically changed graph. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 1, 2015, pp. 69-96. DOI: 10.15514/ISPRAS2015-27(1)-5 (in Russian).

[14]. Burdonov I.B., Kossatchev A.S. Parallel Calculations on Dynamic Graph. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 2, 2015, pp. 189-220. DOI: 10.15514/ISPRAS-2015-27(2)-12 (in Russian).

[15]. Burdonov I.B., Kossatchev A.S. Analysis of directed graph by a set of unmoving automata. Programmnaya inzheneriya [Software Engineering], vol. 8, № 1, pp. 16-25 (in Russian)

[16]. Bourdonov I.B. Traversal of an Unknown Directed Graph by a Finite Robot. Programming and Computer Software, vol. 30, № 4, 2004, pp. 188-203. DOI: 10.1023/B:PACS.0000036417.58183.64
[17]. Bourdonov I.B. Backtracking Problem in the Traversal of an Unknown Directed Graph by a Finite Robot. Programming and Computer Software, vol. 30, № 6, 2004, pp. 305-322. DOI: 10.1023/B:PACS.0000049509.66710.3a
[18]. Lynch, N.A. Distributed algorithms. The Morgan Kaufmann Series in Data Management Systems. Kaufmann, San Francisco, Calif., 1996
[19]. Peleg D. Distributed computing – A Locality-sensitive approach. SIAM Monographs on Discrete Mathematics and Applications. 2000
[20]. Demetrescu C., Finocchi I., and Italiano G.F. Dynamic Graphs. In Handbook of Data Structures and Applications, sec. 36, 2004
[21]. Kushnirenko A.G., Lebedev G.V. Programming for mathematicians. Nauka, Glavnaya redaktsiya fiziko-matematicheskoi literatury [Science, the main edition of physical and mathematical literature], Moscow, 1988 (in Russian)
[22]. Tary G. Le probl`eme des labyrinthes. Nouv Ann Math 14, 1895.

Асинхронные распределенные алгоритмы на статических и динамических ориентированных корневых графах

[1]И.Б. Бурдонов <igor@ispras.ru>
[1]А.С. Косачев <kos@ispras.ru>
[1,2,4]В.В. Кулямин <kuliamin@ispras.ru>
[1,2]А.Н. Томилин <tom@ispras.ru>
[1,3]В.З. Шнитман <vzs@ispras.ru>
[1]Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, дом 25
[2]Московский государственный университет имени М.В. Ломоносова, 119991 ГСП-1 Москва, Ленинские горы, МГУ имени М.В. Ломоносова, 2-й учебный корпус, факультет ВМК
[3]Московский физико-технический институт, 141700, Московская область, г. Долгопрудный, Институтский пер., 9
[4]Национальный исследовательский университет «Высшая школа экономики» 101000, Россия, г. Москва, ул. Мясницкая, д. 20

Аннотация. Эта статья представляет собой обзор серии работ авторов, посвящённых исследованию распределенных систем. Рассматривается асинхронная модель распределённой системы, представленную сильно связанным ориентированным корневым графом, с ограниченной емкостью дуги (в том смысле, что только ограниченное количество сообщений может быть отправлено по дуге за определенный интервал времени). Граф может быть статическим или динамическим, т.е. меняющимся во времени. Для статического графа предлагается алгоритм построения

прямого и обратного остовных деревьев с оценкой времени $\boldsymbol{O}(n / k + d)$, размером памяти в вершине и сообщения $\boldsymbol{O}(n\, d\, log\, \Delta^+)$, где n – число вершин графа, k – емкость дуги, d – длина максимального пути, Δ^+ – максимальная полустепень исхода вершин. Построенные остовные деревья используются в распределенном алгоритме вычисления функции от мультимножества значений, приписанных вершинам графа, за время не более $3d$. В динамическом графе предполагается, что $k = 1$, дуга может появляться, исчезать или менять свой конец. Предлагается алгоритм мониторинга динамического графа, который доставляет в корень информацию о каждом изменении в графе за время $\boldsymbol{O}(n)$ или $\boldsymbol{O}(d)$ после прекращения изменений. Также предлагается алгоритм сбора информации о вершинах графа и разметки графа за время $\boldsymbol{O}(n)$. Эта разметка используется в алгоритме вычисления функции от мультимножества на динамическом графе за время $\boldsymbol{O}(n^2)$ с размером сообщения $\boldsymbol{O}(log\, n)$ или за время $\boldsymbol{O}(n)$ с размером сообщения $\boldsymbol{O}(n\, log\, n)$.

Ключевые слова: распределенные алгоритмы; асинхронные системы; ориентированный граф; корневой граф; динамический граф; параллельные вычисления

DOI: 10.15514/ISPRAS-2018-30(1)-5

Для цитирования: Бурдонов И.Б., Косачев А.С., Кулямин В.В., Томилин А.Н., Шнитман В.З. Асинхронные распределенные алгоритмы на статических и динамических ориентированных корневых графах. Труды ИСП РАН, том 30, вып. 1, 2018 г., стр. 69-88. DOI: 10.15514/ISPRAS-2018-30(1)-5

Список литературы

[1]. Valmir C. Barbosa. An introduction to distributed algorithms. // MIT Press, Cambridge, MA, USA, 1996
[2]. A.D. Kshemkalyani, M. Singhal. Distributed Computing: Principles, Algorithms, and Systems. Cambridge University Press, March 2011. 756 pages
[3]. Michel Raynal. Distributed Algorithms for Message-Passing Systems. Springer Publishing Company, Incorporated, 2013. 500 pages
[4]. Fred B. Schneider. The State Machine Approach. A Tutorial. Fault-Tolerant Distributed Computing. LNCS 448, 1990, pp. 18-41
[5]. Бурдонов И.Б., Косачев А.С. Тестирование системы автоматов. Труды ИСП РАН, том 28, вып. 1, 2016 г., стр. 103-130. DOI: 10.15514/ISPRAS2016-28(1)-7
[6]. Бурдонов И.Б., Косачев А.С. Система автоматов: композиция по графу связей. Труды ИСП РАН, том 28, вып. 1, 2016 г., стр. 131-150. DOI: 10.15514/ISPRAS-2016-28(1)-8
[7]. Бурдонов И.Б., Косачев А.С. Система автоматов: условия детерминизма и тестирование. Труды ИСП РАН, том 28, вып. 1, 2016 г., стр. 151-184. DOI: 10.15514/ISPRAS-2016-28(1)-9
[8]. Бурдонов И.Б., Косачев А.С. Тестирование системы автоматов. Вестник Томского государственного университета. Управление, вычислительная техника и информатика, №1, 2017 г., стр. 67-75.
[9]. Бурдонов И.Б., Косачев А.С. Обобщенная модель системы автоматов. Вестник Томского государственного университета. Управление, вычислительная техника и информатика, №4(37), 2016 г., стр. 89-97.

[10]. Burdonov I.B., Kossatchev A.S. Buidling direct and back spanning trees by automata on a graph. Trudy ISP RAN/Proc. ISP RAS, vol. 26, issue 6, 2014 г., pp. 57-62. DOI: 10.15514/ISPRAS-2014-26(6)-4

[11]. Бурдонов И.Б., Косачев А.С., В.В. Кулямин. Параллельные вычисления на графе. Программирование, том 41, № 1, 2015 г., стр. 3-20.

[12]. Burdonov I.B., Kossatchev A.S., Kuliamin V.V. Parallel calculations by automata on direct and back spanning trees of a graph. Trudy ISP RAN/Proc. ISP RAS, vol. 26, issue 6, 2014 г., pp 63-66. DOI: 10.15514/ISPRAS-2014-26(6)-5

[13]. Бурдонов И.Б., Косачев А.С. Мониторинг динамически меняющегося графа. Труды ИСП РАН, том 27, вып. 1, 2015 г., стр. 69-96. DOI: 10.15514/ISPRAS2015-27(1)-5

[14]. Бурдонов И.Б., Косачев А.С. Параллельные вычисления на динамически меняющемся графе. Труды ИСП РАН, том 27, вып. 2, 2015 г., стр. 189-220. DOI: 10.15514/ISPRAS-2015-27(2)-12

[15]. Бурдонов И.Б., Косачев А.С. Исследование ориентированного графа коллективом неподвижных автоматов. Программная инженерия, том 8, № 1, 2017 г., стр. 16-25

[16]. Бурдонов И.Б. Обход неизвестного ориентированного графа конечным роботом. Программирование, том 30, №4, 2004 г., стр.11-34.

[17]. Бурдонов И.Б. Проблема отката по дереву при обходе неизвестного ориентированного графа конечным роботом. Программирование, том 30, №6, 2004, стр.6-29.

[18]. Lynch, Nancy A.: Distributed algorithms. The Morgan Kaufmann Series in Data Management Systems. Kaufmann, San Francisco, Calif., 1996, pp. 904. ISBN 1-55860-348-4.

[19]. David Peleg. Distributed computing — A Locality-sensitive approach. SIAM Monographs on Discrete Mathematics and Applications. 2000, 359 pp.

[20]. Camil Demetrescu, Irene Finocchi, and Giuseppe F. Italiano. Dynamic Graphs. In Handbook of Data Structures and Applications. Oct 2004, 36-1 -36-20. ISBN: 978-1-58488-435-4.

[21]. Кушниренко А.Г., Лебедев Г.В. Программирование для математиков, Наука, Главная редакция физико-математической литературы, Москва, 1988.

[22]. Tary G. Le probl`eme des labyrinthes. Nouv Ann Math 14, 1895.