Iurii Petrov

IMPLEMENTING GRAPH REPRESENTATION MODEL FOR PARALLEL AND DISTRIBUTED SYSTEMS USING ERLANG

Abstract *This paper describes a new model of graph representation using the Erlang actor model. Benefits of using lightweight processes instead of traditional passive data structures are given. Examples of using this model are shown. The experimental part gives two examples of using early implementations of the offered model. As a conclusion, an analysis of applicability for tasks from different branches of science is given.*

Keywords graphs representation, parallel systems, distributed systems, clusters, Erlang

Citation Computer Science 17 (1) 2016: 99–120

1. Introduction

Since the early '80s, there has been a problem of distributing big graphs. To date, there are problems like:

- load balancing – graph structure can be changed during lifetime of model;
- representing graphs in heterogeneous systems;
- multi-user problem – sometimes it is theoretically possible to run two or more tasks simultaneously, but because all traditional models are synchronous, tasks usually have to wait for their turn to start, this problem reduces productivity of distributed systems.

Most modern graph representation models are still synchronous; this means that they can alleviate these problems but not solve them. Probably the main reason is the legacy code. These models share the idea of representing vertices as some passive data structures, since representing graphs as adjacency matrices in the '80s [3, 6, 8]. Computing systems have changed dramatically over the last 30 years, so there should be new rules for creating graph representation models:

1. virtualization is a modern trend, and the current computer hardware is ready for it. Virtualization has many benefits and can solve some problems; in the case of cluster systems, it is a problem of heterogeneous systems: it can be very difficult or even imposible to represent data structures identically in systemswith different architecture;
2. asynchronous models instead of synchronous – because a synchronous model can be made out of an asynchronous, but the reverse is not true;
3. using processes instead of passive data structures; this means vertices are actors, in consideration of previous rule, there is no need to wait while one task will finish before starting another one;
4. hierarchy for horizontal scalability.

Some of these rules are already mentioned in some graph models; e.g., PowerGraph [4]. This system also uses the idea of active structures (in PowerGraph, they are called "vertex-programs"), but this system is probably not suitable for a heterogeneous environment.

The ideal language for implementing such a model is Erlang. Its other benefits are perfect scalability and fault tolerance, which will also be used in the offered model. Further, the model is named "asynchronous graph representation model". Note that the model is parallel by its nature, but realization is concurrent because of using Erlang.

2. Notations used

For processes, standard Erlang designations are used [1]. Erlang notation is used for Erlang data types, such as lists ([ ]), tuples ({ }), atoms (' '), and strings (" "). Variables start with a capital letter, and names of tables are atoms. Traditional

Erlang notation for sending messages is used (!). For illustrating connections between processes, different types of lines and arrows are used. Note that the arrows show the direction of message passing, but connections between communicator and its vertices are the exception: because a communicator can be the parent of thousands of processes, the arrow from the vertex to the parent communicator is not shown (but the vertex can send messages to its parent communicator). Graphical notations are presented on Figure 1.

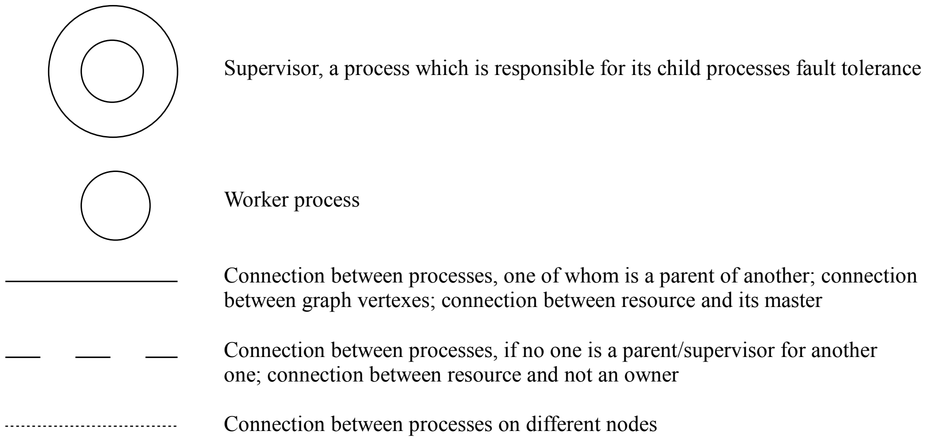

Supervisor, a process which is responsible for its child processes fault tolerance
Worker process
Connection between processes, one of whom is a parent of another; connection between graph vertexes; connection between resource and its master
Connection between processes, if no one is a parent/supervisor for another one; connection between resource and not an owner
Connection between processes on different nodes

Figure 1. Graphical notations.

There are a few ways of marking an edge (if necessary) by writing one of these notions on the line:

- EID, Weight – when the edge is represented by both its name (edge ID; e.g., 'u1') and weight;
- EID – when the edge is represented only by its name;
- Weight – when the edge is represented only by its weight.

Edge ID (EID) is recommended to be a symbolic name (atom). Only one way of marking edges should be used.

3. Model's hierarchy

The described model is hierarchical. It contains 3 levels:

1. Global managing level – level for interaction with user and manipulating global data.
2. Local managing level – level for computations and manipulating local data.
3. Workers' level – vertices.

Figure 2 demonstrates the model hierarchy.

Global data is graph allocation and information about nodes. This means that any process on the global managing level cannot send a message directly to the vertex.

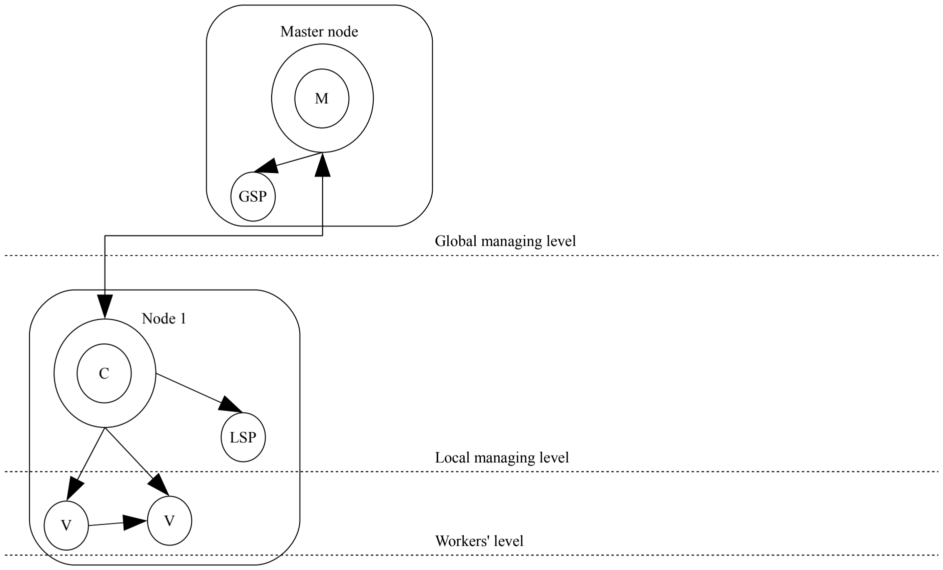


Figure 2. Model hierarchy.

It must send a message to the vertex's parent communicator first, then the parent communicator will pass this message on to the vertex. From the other side, processes on local managing level usually do not have any information about processes on the other computing nodes. Communicators are the exclusion; they have information about each other's unique IDs and number of vertices connected to the other nodes (see "Communicator" further), but they do not have information about the other communicators' PIDs, LSPs, or vertices on the other nodes. Processes on the workers' level do not have any information even about the other processes on the same node (unless they are directly connected).

This hierarchy is good for fault tolerance: supervisors isolate errors and crashes on lower levels, so if one node crashes, the whole system will still work and the crashed node will be respawned by the master process. A special section further is dedicated to mechanisms of fault tolerance.

4. Types of processes in the model

Short definitions of processes are provied in the glossary (at the end of the article). This part of the article is dedicated to the processes' responsibilities and their cooperation.

4.1. Required processes

This subsection describes the required entities to create a distributed graph.

4.1.1. Master process

Master process (M) is the root process of hierarchy. It is responsible for the creation of all of the required processes: communicators, transaction server. It is also responsible for their fault tolerance. There can be only one master, but it can be changed manually (see further). The master process owns at least one DETS and at least two ETS tables. They are required to be named tables:

- 'graph' (filename: graph.db) – DETS table, type: set. Contains basic information about the allocated graph: vertices, edges (optionally), weights of edges (optionally);
- 'nodes' – ETS: type: set. Contains associations between nodes' symbolic names (defaults are numbers) and PIDs of their communicators;
- 'vertices' – ETS, type: set. Contains associations between vertices (represented by their IDs) and nodes on which these vertices are allocated.

Figure 3 demonstrates the place of the master process in the hierarchy as well as its resources.

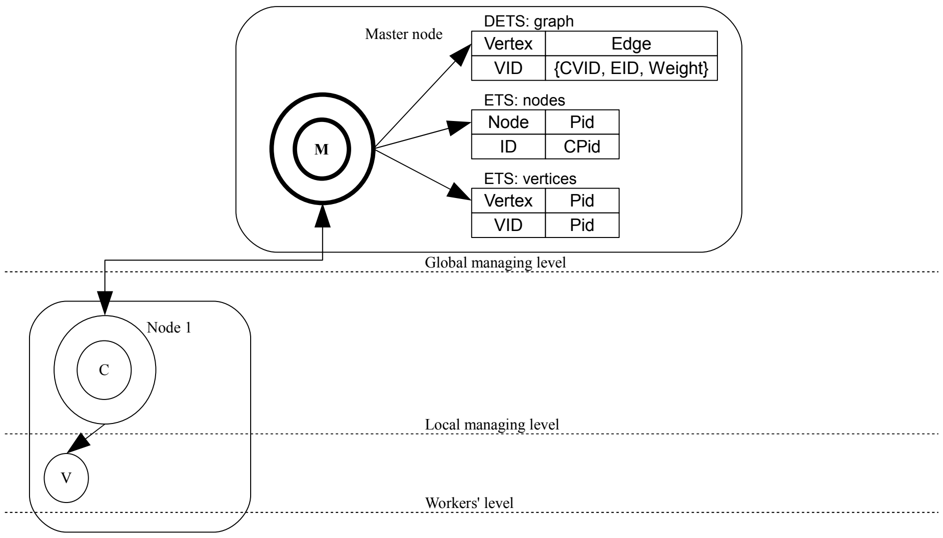


Figure 3. Master process and its resources.

4.1.2. Communicator

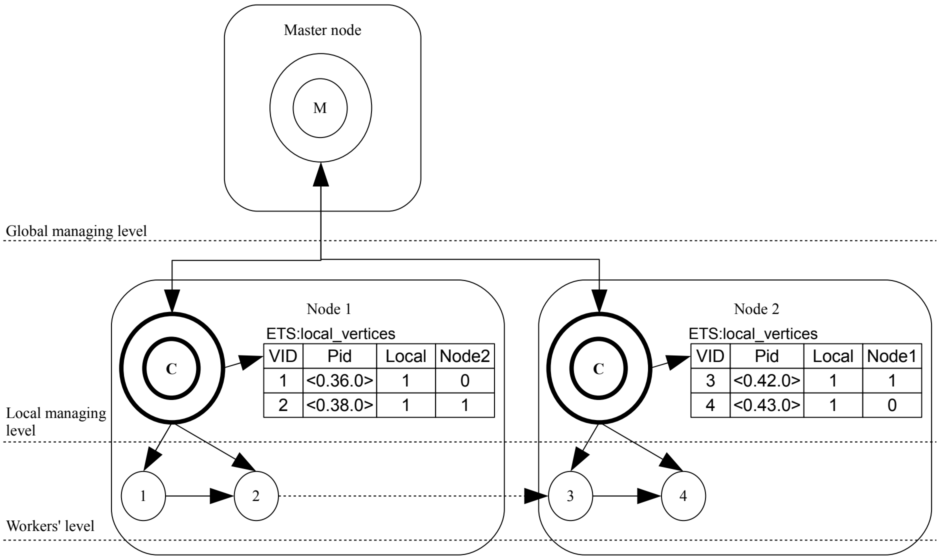
Communicator (C) is the root process of computing node, responsible for the creation of the vertices' processes and their fault tolerance by respawning them if they crash. Optionally, they can spawn LSP processes (it is recommended for the communicator to be a supervisor for its LSPs). Communicator by default owns only one named ETS table: 'local_vertices'. Type of the table is set, it contains the following columns:

- VID – vertex unique ID, same as in master's vertices table, of allocated vertex;

- Pid – process ID of vertex's process;
- Local – number of connections between the vertex with the other vertices on this node;
- N-1 columns entitled like "NodeM", there N is a total number of nodes, M is a name of another node; this cell contains the number of connections with vertices on node M.

E.g., if there are 3 nodes in the system, 'local_vertices' table's columns of node number 2 will look like {VID, Pid, Local, Node1, Node3}. The number of connections is a basic condition for load rebalancing by migrating the vertex having more connections with some distributed node than to local vertices (about possible load balancing mechanisms, see the special section).

Figure 4 demonstrates the place of the communicator in the hierarchy and its resources.



VID	Pid	Local	Node2
1	<0.36.0>	1	0
2	<0.38.0>	1	1

VID	Pid	Local	Node1
3	<0.42.0>	1	1
4	<0.43.0>	1	0

Figure 4. Example of communicators and their resources.

4.1.3. Vertex

Vertex (V) is a simple worker process, it does not own any resources, but it has its state described by:

- VID – vertex unique ID;
- CPid – PID of parent communicator;
- list of current tasks (LoT) – contains tasks IDs (TIDs) of tasks, in which the vertex is involved now (see description of TS for understanding);

- list of connections (LoC) – list, containing tuples like {CVID, Pid, EID, Weight} where:
  - CVID (connected vertex ID) is unique ID of the vertex, to which an outgoing connection is going,
  - Pid – PID of CVID vertex process on a local or remote node,
  - EID – unique ID of edge, 'u1' by default,
  - Weight – weight of the edge, 1 by default;
- list of Neighbors (LoN) – same as LoC, incoming connections instead of outgoing (NVID instead of CVID), in an undirected graph this list is empty.

This state is used in the loop function of the vertex process. The structure of the loop function's stack can be modified in some cases (like migration).

Figure 2 demonstrates the place of the vertex process in the hierarchy.

4.1.4. Transaction server

Transaction server (TS) is another required process. It always exists on the master node. It owns one ETS named table 'tasks.' Record in this table looks like {TID, Operands, Task, Status}, where:

- TID – unique task ID, an integer;
- Operands – list of communicators' PIDs of nodes, there operands are allocated; in case of operation upon the whole graph, this list contains all the nodes;
- Task – task's short description, atom (e.g., 'maxweight');
- Status – list (empty or containing nodes, on which task has been completed) or atom 'ok' (if Status becomes equal to Operands).

TS algorithm:

1. User sends a command to master by invoking one of its interface functions.
2. Master process sends a message to TS.
3. TS writes it to 'tasks' table, then sends the task to all the nodes in Operands.
4. Communicators pass the task to their nodes.
5. Vertices add task to their LoTs and begin execution.
6. After completing the task vertex passes results to its parent communicator (or LSP, if set explicitly).
7. Communicator passes results to TS (or GSP, if set explicitly).
8. TS adds the node's symbolic name to Status.
9. If Status contains all the nodes from Operands, Status='ok', Operands ! {TID, Completed}.
10. Communicators send TID, completed to their vertices.
11. Vertices exclude TID from their LoTs.

This is a coarse-grained style of computation, but sometimes it is necessary to do some simple task affecting only few vertices. In this case, Operands and Status will contain tuples {Cpid, VID}; there, Cpid is a communicator's PID (from 'nodes' table), VID is a vertex's unique ID. Communicator will also send vertices' VIDs in the response message.

TS is a registered process named 'TS,' which means the master or any other process on the managing node does not need to know its PID to send messages. If support masters are in use, TS owns the named ETS table 'TSS' where PIDs of TSS processes are contained.

TS is designed for fault tolerance. If one of the nodes crashes before completing its task, the task can be restarted for the respawned node. Note that by default operations, affecting the original graph topology, are not processed by TS. These operations are executed directly because they are assumed never fail and crash a node, which means they do not need supervising. This can be fixed if a full change log is needed. The reverse is also true: some fast and secure operations can be performed directly, without TS.

Figure 5 demonstrates the place of TS in the hierarchy and its resources.

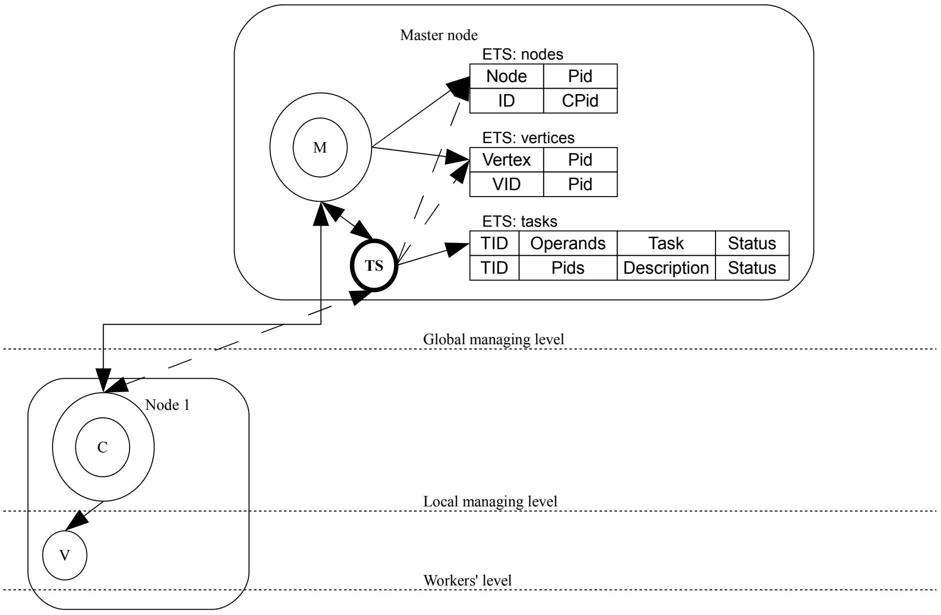


Figure 5. Transaction server and its resources.

The master process and TS are the only two basic required processes. For allocating graph and computing, there should be at least one node. The master node and computing node can be the same node, but such a configuration is not recommended. So in a working system there could be only four types of processes: master, transaction server, communicator, and vertex. These processes form a required processes group. Figure 6 demonstrates a simplified situation when a small graph is partitioned between two nodes, and only the required processes are used.

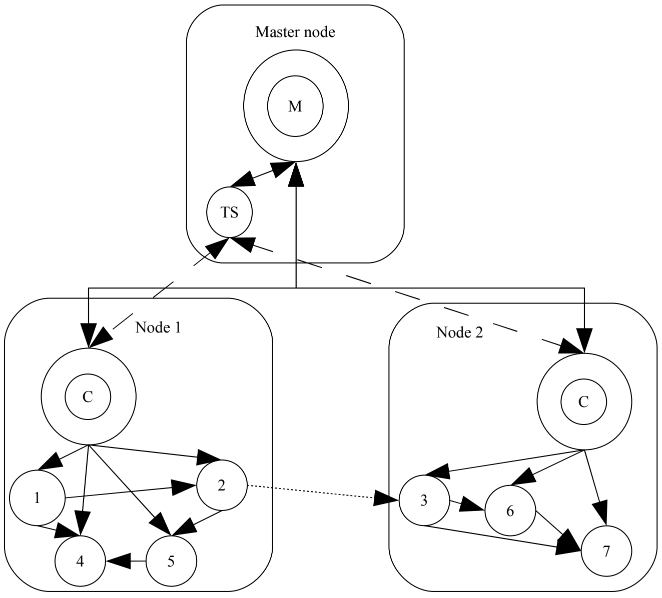


Figure 6. Basic model.

4.2. Support processes

The other group of processes is called support processes. There are two special and two general types of processes in this group. These processes are used for extending the functionality of the model and additional fault tolerance.

4.2.1. Support master process

The support master process (SM) is a special type of support process. It is designed for both multi-user support and fault tolerance. There are two ways of creating an SM process: by spawning from the master node and by creating it manually and then adding to a master node. In the first case, SM is created by calling the interface function, arguments are the address of remote node and symbolic name of SM. In the second case, the remote node should contain compiled sources of the SM module. The user manually calls the SM creation function; after creation, the SM process should be attached to the master node by invoking a special function, which arguments are PID of master process and symbolic name of SM process. Master process can accept or reject registration of the SM process. If a symbolic name is already in use, the master process rejects registration of the SM and sends back a reply {reject, name_in_use}, SM can be registered with another name. Default names are strings like “SM1” etc. After successful registration, SM receives all required information from master.

SM owns copies of master process tables. If the master or any SM process makes changes, it sends a message with the changes to all of the other owners of the same tables. Such design is a key feature for fault tolerance (see the special section about it).

SM can do most of the master process's functions, but it is not responsible for fault tolerance of the other nodes. The master can be responsible for SM fault tolerance. The master process is responsible for SM fault tolerance in case the SM was created by the master (supervision can be disabled if explicitly set). Supervision can be added later by invoking a special function of the master or SM process. Figure 7 demonstrates place of support master process in hierarchy and its resources.

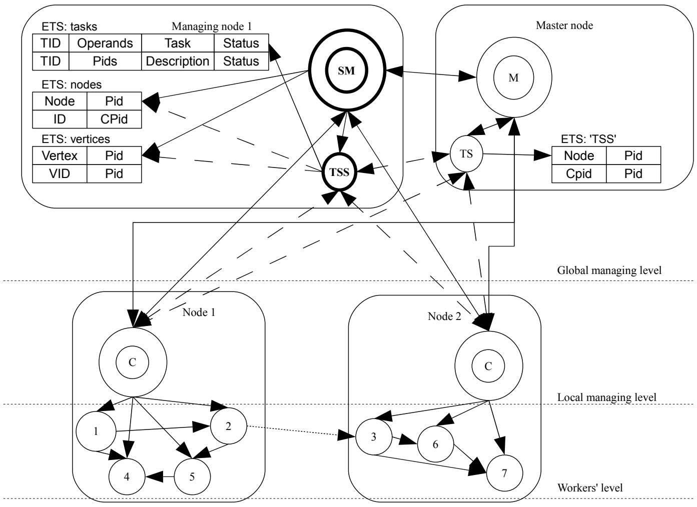


Figure 7. SM, TSS and their resources.

4.2.2. Support transaction server

The support transaction server (TSS) is another special type of support process. Like the SM process, it owns a copy of TS's table. TSS is connected to TS. The mechanism of connection is similar to M-SM, after establishing a connection between M and SM, TSS sends its PID to TS, TS adds it to the 'TSS' table. TSS is automatically created after the creation of SM. If the user wants to execute a task using SM, SM passes the task to its TSS. Then, TSS assigns TID to this task (based on its copy of 'tasks' table) and sends a message like {TID, Operands, Task} to TS. If no such TID is in use, TS sends {task, TID, accept} message to TSS and then {task, register, {TID, Operands, Task}} to all of the other TSSs. If TID is already in use, TS sends back a message {task, TID, reject}, then TSS tries to assign another TID and register the task again. After the task is registered, TSS sends it to Operands. Note that the logic of the model is flexible, so there could be variations of TIDs generating mechanism; e.g., TS can reserve and send some diapason of TIDs to TSSs.

Figure 7 demonstrates the place of the support transaction server and its resources.

4.2.3. Global and local support processes

There are also two types of generalized support processes – GSP and LSP. GSP and LSP can be called "service processes".

GSP stands for global support process. It can be any process created on the global managing level. In most cases, it should have access to global data; but this is a recommendation, not a restriction. GSP can be, for example, a process providing a web command line interface for sending commands.

LSP stands for local support process. It can be any process created on the local managing level. They are similar to GSPs.

GSPs and LSPs can spawn their own child processes: GSPs and LSPs respectively. A good example of using GSPs and LSPs is a reduction of data, for example, searching for a maximum weight of an edge:

1. User sends command via master process.
2. Master process creates GSP for global data reduction.
3. GSP passes message with its PID and task to TS.
4. TS sends the task to all the nodes.
5. Each communicator creates LSP on its node.
6. LSPs send the task to vertices on their nodes and their PIDs.
7. After completing task, vertex sends result to LSP.
8. When all vertices are processed, LSP sends result with local maximum to communicator.
9. Communicator sends messages to TS and GSP.
10. GSP waits for all nodes to send their local result, then finds a global maximum.
11. GSP sends global maximum to master process.

Note that this algorithm is generalized; some details (like how LSP can count vertices on its node) are not given, but this situation is a good example of data reduction with GSP and LSP: due to these, the master process and communicators are free of computations and ready to receive and pass tasks, thus increasing the utilization of resources.

Figure 8 demonstrates examples of GSPs and LSPs and their interaction.

4.2.4. Restrictions for support processes and other types of support processes

All support processes have only two restrictions:

1. They should not decrease fault tolerance of the whole system.
2. Support processes should be on their places. That means spawning web CLI on the global managing level is normal, but spawning the same process on the local managing level is not.

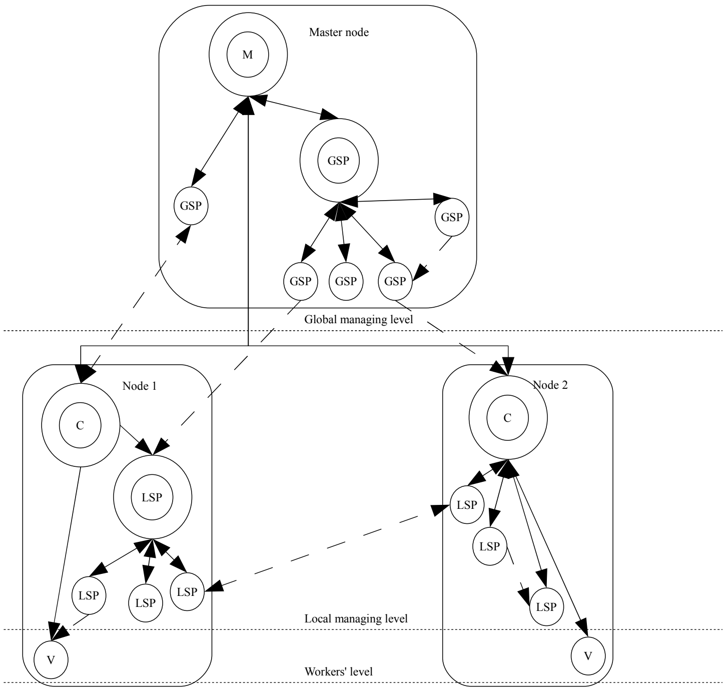


Figure 8. Examples of GSPs and LSPs.

Search agent (SA) is a special type of GSP, but it is a temporary process that will be described further (see “The initial allocation task”). Any other new support processes can be implemented as GSPs or LSPs.

5. Model’s fault tolerance

Using Erlang always means a positive side effect: fault tolerance. The model designed in compliance with all basic Erlang requirements. Also, some additional features are implemented. The basic level of fault tolerance is isolating errors on the lower levels. Because of this requirement, the master process and communicators are supervisors. It does not matter which process has crashed – it can be respawned.

TS is a process designed for fault tolerance. If one of the nodes has fallen during execution of one or a few tasks, the task can be restarted only for this node. This means the results of computation will not be lost in case of a few or even all nodes crashing. For example, four of nine nodes processed their task, and then all nine nodes crashed. The master node will respawn these nodes and send their symbolic names and PIDs to TS. TS will find which nodes had not processed the task before they crashed and restart the task only for those five nodes.

The other way to achieve fault tolerance is reservation. Support masters are not only designed for multi-user support, but also for reservation. In case the original master node has crashed, one of the support masters has to become the new master. It should send to other SMs message {new_master, PID} with its own PID and then become a supervisor for the nodes. Note that GSPs on the master node will not be respawned by default. They can be respawned if the master process owns an ETS table with the description of running GSPs (e.g., {PID, Module, Arguments}), and copies of this table are synchronized with SMs.

6. Model's load balancing

Another problem of traditional models is load balancing. Because of representing vertices as passive data structures, it is hard to reallocate vertices during computations. In the asynchronous parallel graph representation model, vertices can do the majority of the load balancing job. The basic algorithm for load balancing:

1. Communicator (or LSP) periodically checks the vertices table. If it finds that some vertex has more connections with any other node – communicator sends a message for allocation of a new vertex.
2. Remote communicator receives the message, if it is possible to spawn a new process for the vertex (number of vertices per node can be restriction) remote communicator spawns a new process and sends back approval with PID.
3. If the communicator receives acceptance, it passes the message of reallocation to the vertex to be reallocated.
4. The vertex process enters a state of migration. It sends to all vertices from LoT and LoC its new PID on the remote node and symbolic name of its new communicator, all received messages are being redirected to the new vertex process. Communicators of connected processes update their vertices tables.
5. The vertex process sends its state (VID, LoT, LoC, LoN) to the new process.
6. The new process registers in the remote node's 'local_vertices' table, then sends a message to the original vertex process.
7. The original vertex process sends a message to its communicator, which deletes the record of the vertex from its 'local_vertices' table, then the process is terminated by itself.
8. Concurrently with Step 7, the communicator of the new vertex process sends to the master a message about reallocation. Master updates its 'vertices' table.

This is just a basic algorithm which can be modified. In accordance with peculiar properties of each graph, the conditions of rebalancing can be changed.

7. The initial allocation task

For initial allocation, there is a special temporal GSP – search agent. The search agent is a flexible component, so it does not have such explicitly described logic as a TS or any other process. It is claimed to make the initial allocation of vertices and initial load balance. A generalized algorithm:

1. Master process creates SA and sends to nodes its PID and task of initial allocation.
2. Nodes send messages to SA to get a "portion" of vertices to allocate. "Portion" is a group of vertices found in graph.db, having minimum connections with the other nodes. This is the most flexible part of SA logic – any of the algorithms of the partitioning graph can be used.
3. SA sends a "portion" of vertices to the node and message about allocation to the master process. When all of the vertices are allocated, SA sends a message to the master process and terminates itself.

Options for SA:

- algorithm used for finding vertices ([7] gives some applicable algorithms);
- number of vertices in "portion";
- number of SAs; by default it is 1, but can be changed;
- support processes for SA; bone by default;
- method of marking allocated vertices; it can be an ETS copy of graph.db, records are being deleted during allocation.

The algorithm of initial allocation should not be very strict if it decreases performance. After initial allocation, a standard load-balancing algorithm starts to work, so some lack of initial allocation will be fixed shortly, while the graph will be available for executing tasks.

8. Experimental part

This section describes results of the experiments on two early implementations of the asynchronous graph representation model. For chronometry, standard Erlang function `timer:tc/3` was used, granularity is up to 10 milliseconds.

8.1. Searching for the maximum weight

The first experiment is searching for the maximum weight among edges. Used algorithm:

1. Master receives request for the edge of maximum weight.
2. Master spawns GSP for data reduction, arguments are PID of master and number of computing nodes.
3. Master sends request to communicators with additional argument: PID of GSP.
4. Communicators receive message from master process and spawn their own LSPs, arguments are their own PIDs and number of vertices on their nodes.
5. Each communicator sends message maxweight, ServicePid to one of its vertices, ServicePid is a PID of LSP.

6. Vertex receives messages, finds maximum among its edges, sends it to LSP, and then passes the message to all vertices from LoC. Vertex blocks itself for receiving messages with this task.
7. LSP receives a message, checks if weight greater than previous maximum; if it is – LSP updates information about the edge with maximum weight. LSP receives messages until the number of received messages is not equal to the number of vertices per node.
8. After all vertices on the node are processed, LSP sends the local maximum to the communicator.
9. Communicator passes message to GSP.
10. GSP's logic is similar to LSP's on step 7: GSP compares weight in received message with temporary global maximum and updates temporary global maximum, if needed.
11. After all nodes are processed, GSP sends result to the master.

Note that this algorithm can be modified (e.g., Step 9 is not necessary, LSP can pass its messages directly to GSP). Utilization of resources depends on the number of connections per vertex, how many vertices received a message on Step 6 (in the algorithm above – just one vertex). For the experiments we also used an algorithm that did not have these limitations (the number of vertices on Step 6 was two, Step 9 excluded).

Listing 1. Part of LSP module.

```
%% all vertices are processed
loop(Master, Size, Size, MWeight) ->
 Master ! {weight, MWeight};

%% Master = Pid of parent communicator,
%% Size = number of node's allocated vertices
%% Processed = number of processed vertices, MWeight = maximum weight
loop(Master, Size, Processed, MWeight) ->
 receive
  {ID, Weight} ->
   case Weight > MWeight of
    true -> loop(Master, Size, Processed+1, Weight);
    _    -> loop(Master, Size, Processed+1, MWeight)
   end
 end.
```

Listing 2. Part of Vertex's loop code.

```
%% This implementation does not use TS yet, instead it uses State
    ↪ variable
%% State = 0 -- vertex is free for tasks, State = 1 -- vertex is locked
loop(Master, ID, ListOfConnections, ListOfNeighbors, State) ->
 receive
  {maxweight, ServicePid} ->
   case State of
```

```
    0 -> Weights = [Weight || {_CID, _CPID, Weight} <-
    ↪ ListOfConnections],
         MaxWeight = lists:max(Weights),
         ServicePid ! {ID, MaxWeight},
         lists:foreach(fun(Element) ->
          {_CID, CPID, _EWeight} = Element,
          CPID ! {maxweight, ServicePid}
         end,
         ListOfConnections);
    1 -> ok
   end,
  loop(Master, ID, ListOfConnections, ListOfNeighbors, 1)
 end.
```

In C++ project graphs were represented as adjacency matrices.

Listing 3. C++ code of searching path.

```
start = omp_get_wtime();
#pragma omp parallel for
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
   localmax = (localmax > M[i][j])?localmax:M[i][j];
finish = omp_get_wtime();
```

Note that this code contains a race condition (variable `localmax`) because Microsoft Visual Studio 2008 does not support reduction operator `max` . Weights of edges were between 1 and 10.

There were several types of experiments. The first type was Erlang R15B03 and one node, unoptimized version of algorithm. The second type was Erlang R15B03, one node, optimized algorithm. The third type was Erlang R15B03, graph was partitioned between two local nodes (on the same SMP system), only for 500 and 1000 vertices. The fourth type was similar to the third, but one of the nodes was remote (small Beowulf cluster). The fifth type was Erlang R17.3, unoptimized algorithm. All scheduler settings were default. For C++ experiments, accuracy is about 1 microsecond (according to `omp_get_wtick()`).

In most types of experiments, the following numbers of vertices were used: 10, 20, 40, 50, 100, 200, 250, 500, and 1000. For each number of vertices, 100 experiments were performed. The used graph was representing some electrical chain (Figure 9). It is clearly seen from the figure that vertices 1, 6, etc. can be bottlenecks.

Specifications of used nodes:

- Desktop computer Sony VAIO VPCL13S1R (local node):
  - CPU: Intel Core2Quad Q84000, 2,66 GHz;
  - RAM: 8 GB, DDR2;
  - Windows 7 Home Extended, 64-bit.
- Laptop computer ASUS ZenBook UX51A (remote node):
  - CPU: Intel Core i7 3517U, 1,9 GHz (2 physical cores + hyper-threading);

– RAM: 4 GB, DDR3;
– Windows 7 Professional, 64-bit.

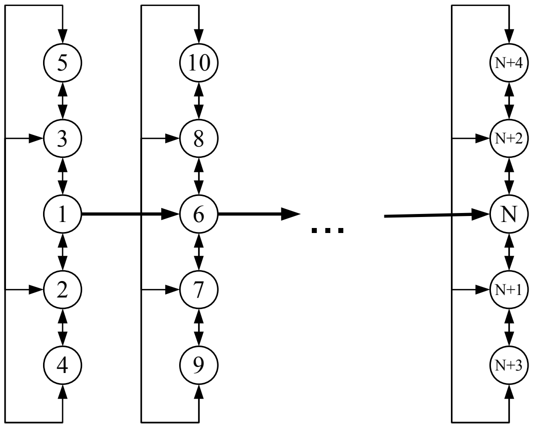


Figure 9. Structure of experimental graph.

Connection between remote nodes was established via Wi-Fi (802.11g). Table 1 describes results of the experiments. To assess the scalability of the model, average times of processing per vertex were calculated (Table 2).

Table 1
Average time of searching the edge of maximum weight.

Platform	Number of vertices	10	20	40	50	100	200	250	500	1000
Erlang/OTP R15B03	Local node, unoptimized algorithm, μs	330	360	770	920	1540	2030	2020	3300	—
	Local node, optimized algorithm, μs	160	310	300	770	1240	2000	1710	3280	—
	Tow local nodes, μs	—	—	—	—	—	—	—	3290	6730
	Local and remote nodes, μs	—	—	—	—	—	—	—	15280	16840
Erlang/OTP R17.3	Local node, unoptimized algorithm, μs	150	300	610	600	920	1100	1700	3300	6890
C++ OpenMP	4 threads, μs	2144	2293	2161	2171	2422	3865	4915	11526	35897

As seen from the tables above, the model proved itself to be highly scalable. Experiments with two nodes proves that the model is suitable for clusters: two local nodes process a task with the same performance. Decreasing performance in the case of two remote nodes is wireless network latency. A graphical interpretation of the results is shown on Figure 10 (only for one node).

Table 2

Time of processing per vertex.

Platform	Number of vertices	10	20	40	50	100	200	250	500	1000
Erlang/OTP R15B03	Local node, unoptimized algorithm, μs	33	18	19,25	18,4	15,4	10,15	8,08	6,6	—
	Local node, optimized algorithm, μs	16	15,5	7,5	15,4	12,4	10	6,84	6,56	—
	Tow local nodes, μs	—	—	—	—	—	—	—	6,58	6,73
	Local and remote nodes, μs	—	—	—	—	—	—	—	30,56	16,84
Erlang/OTP R17.3	Local node, unoptimized algorithm, μs	15	15	15,25	12	9,2	5,5	6,8	6,6	6,89
C++ OpenMP	4 threads, μs	214	115	54	43	24	19	20	23	36

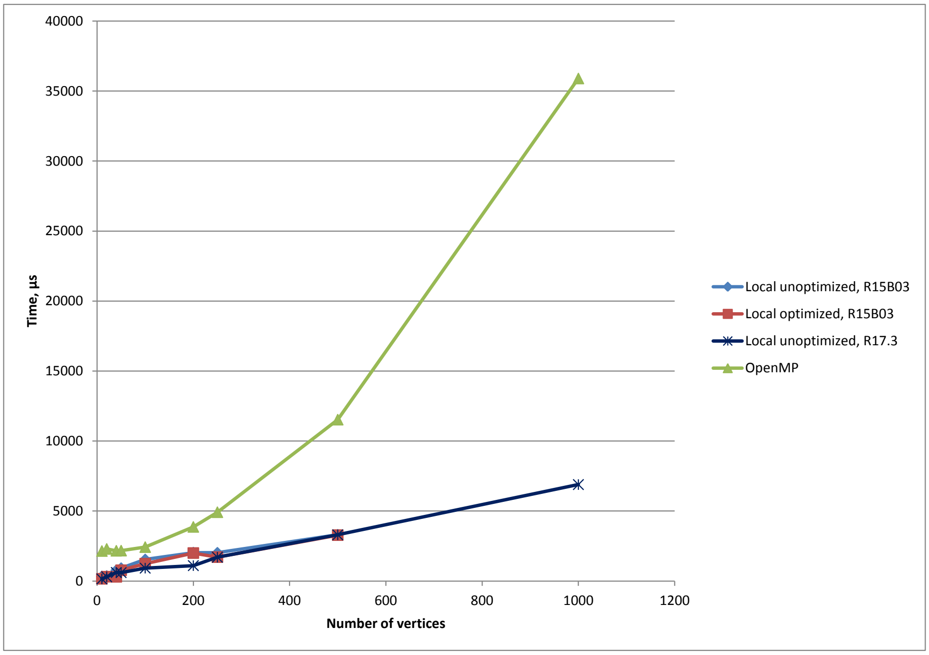


Figure 10. Dependency between number of vertices and time of discovering path.

8.2. Discovering path in De Bruijn graph

Another experiment was searching a path from vertex A to vertex B in a De Bruijn graph. Conditions of the experiment were:

1. Binary alphabet for De Bruijn sequence.
2. Erlang/OTP R17.3.

3. Graph allocated only on one local node (same node as in previous experiment).
4. For eight vertices, paths with lengths of 3 and 4 were found (start and destination vertices are included in path), for 16 – 3, 4 and 5, for 32 – 3, 4, 5 and 6 etc., but not more than four series of experiments for each number of nodes. Longer paths have higher priority.
5. For each length of path, 100 experiments were performed, which gives up to 400 measurements for each number of vertices.

Path search algorithm:

1. Master receives message with IDs of start vertex (1 by default) and destination vertex.
2. Master passes task to communicator, which contains the vertex.
3. Communicator sends message like {path, N, ServicePid, []} to start vertex, where N is a VID of destination vertex, ServicePid is a PID of communicator.
4. Vertex received this message compares its VID and N, if equal – adds its VID to current path (CurPath variable in the message) and sends result, CurPath to ServicePid; if not equal – adds its VID to CurPath and sends message {path, N, ServicePid, CurPath} to all vertices in LoC. After this vertex block itself for receiving messages with tag 'path' to prevent double passage.
5. Communicator receives message with tag 'result' and passes it to master.

Listing 4. Part of Vertex's loop code.

```
loop(Master, ID, ListOfConnections, ListOfNeighbors, State) ->
  receive
    {path, N, ServicePid, CurPath} ->
      case State of
        0 ->
          if (ID == N) ->
            ServicePid ! {first_path, CurPath};
          true           ->
            lists:foreach(fun(Element) ->
              {_CID, CPid}=Element,
              CPid ! {path, N,  ServicePid, CurPath ++ "␣" ++[ID]}
              end,
            ListOfConnections)
          end;
        1 -> ok
      end,
      loop(Master, ID, ListOfConnections, ListOfNeighbors,1)
  end.
```

This algorithm has at least one flaw – it contains a race condition (e.g., when path length 5 exists, the first result could be path length 6). The race condition did not affect the results during our experiment. Results of experiment are given in Table 3.

Table 3

Results of experiments to search paths in De Bruijn Graphs

Time\Number of vertices	8	16	32	64	128	256	512	1024
t_{avg}, μs	0,7	0,05	0,435	0,398	0,545	1,375	2,865	5,802
t_{ver}, μs	0,09	0,003	0,013	0,006	0,004	0,005	0,005	0,005
t_{path}, μs	0,2	0,017	0,09	0,082	0,095	0,187	0,344	0,633

Where:
t_{avg} – average time of searching path;
t_{ver} – average time of processing one vertex;
t_{path} – the most complicated parameter: for each path length local average is calculated as average time of searching path divided by path length, after calculating all these local averages t_{path} is calculated as an average of them, in other words – average time of searching one vertex in the path.

Graphical representation of results is given on Figure 11.

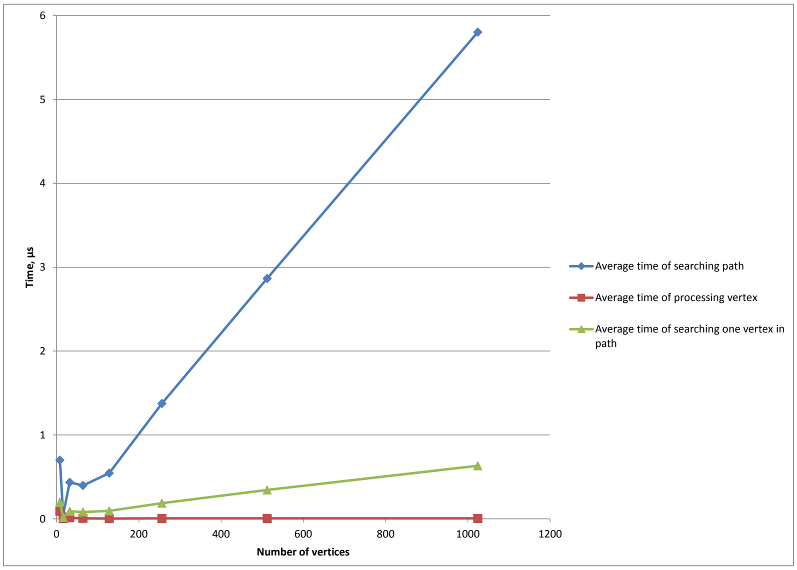


Figure 11. Results of searching paths in De Bruijn graphs.

The model proved itself to be scalable and effective for basic tasks of bioinformatics [2, 5] because of slower than linear growth of main indicators such as time of execution.

9. Conclusion

Experiments on two problems from different branches of science proved that the model is applicable for various types of graphs. Despite the fact that early implementations of the model were used, they showed high scalability. In the future, it is planned to implement all entities of the model and test it on the same experiments with bigger graphs, scalability on larger Beowulf clusters and on enterprise clusters. The model is also applicable for grid computing because of its efficiency in heterogeneous systems (which is provided by using Erlang virtual machines) and fault tolerance for tasks.

10. Glossary

Global managing level – level in the hierarchy, contains processes and resources, which have permissions only to global level, not local (e.g., process on global managing level cannot control process on workers' level directly), can change original graph topology.

Local managing level – level in the hierarchy, contains processes, which can affect on working processes directly, but cannot change the original graph topology (representation topology can be changed on this level).

Workers' level – level in the hierarchy, contains Erlang working processes representing vertices.

Node – Erlang VM, local or remote.

Managing node – node, contains a process, which provides interaction between user and model.

Computing node – node, which only takes part in computations, has no direct interfaces for interaction with user.

Master process (M) – root process of process hierarchy, main process on global managing level, responsible for spawning all required processes on global and local managing level. Provides an interface for interaction between user and model.

Master node – node containing current master process.

Transaction server process (TS) – process on the master node, required to send tasks to communicators and nodes.

Support master process (SM) – process on managing node, provides interface for interaction between user and model, acting as a reserve master process, but cannot spawn processes on local managing level. Can become a master.

Support transaction server process (TSS) – similar to SM, used as a reserve TS, but can become a TS if master node crashed.

Support process – not required process, used for some subsidiary tasks.

Global Support process (GSP) – any support process on global managing level; e.g., data collector from communicators for further reduction and sending to master process.

Local Support process (LSP) – any support process on local managing level, similar to GSP.

Communicator (C) – main process on each computing node, parent of vertices and local support processes on this node.

Vertex (V) – Erlang process, representing vertex of graph, also includes some additional information (see further).

Search agent (SA) – special type of GSP, used for initial allocation of vertices.

Task – any computation initiated by user, which does not change topology of original graph.

References

[1] Cesarini F., Thompson S.: *Erlang Programming*. DMK, Moscow, 2012.

[2] Compeau P., Pevzner P., Tesler G.: How to apply de Bruijn graphs to genome assembly. *Nature Biotechnology*, vol. 29, pp. 987–991, 2011.

[3] Gergel V.P.: *Theory and Practice of Parallel Computing*. BINOM, Moscow, 2007.

[4] Gonzalez J., Low Y., Gu H., Brikson D., Guestrin C.: PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs, `http://www.eecs.harvard.edu/cs261/papers/gonzalez-2012.pdf`.

[5] Homolog.us – bioinformatics: *Why use de Bruijn Graphs for Genome Assembly?* `http://www.homolog.us/Tutorials/index.php?p=1.4&s=1`.

[6] Ovchinnikov V., Ivanova G., Nichushkina T.: Selection of Data Structures for Graph Representantion while Solving Combinatoril and Optimisational Problems. *Herald of the Bauman Moscow State Technical University. Instrument Engineering*, vol. 2 (43), pp. 39–51, 2001.

[7] Ovchinnikov V.A.: *Algorithmization of Combinatory-Opimizational Tasks in Constructing Computing Systems*. Publishing house of Bauman Moscow State Technical University, Moscow, 2001.

[8] Yakobovskiy M.: *Introduction to the Parallel Methods of Problem Solving*. Publishing house of Moscow State University, Moscow, 2013.

Affiliations

Iurii Petrov
Bauman Moscow State Technical University (BMSTU), Moscow, `ykpetrov@bmstu.ru`

Received: 10.02.2015
Revised: 12.04.2015
Accepted: 30.05.2015