SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

DFA-G: A Unified Programming Model for Vertex-centric Parallel Graph Processing

Bo Suo, Jing Su, Qun Chen, Zhanhuai Li, Wei Pan

2016-08-19

1 ABSTRACT

Many systems have been built for vertex-centric parallel graph processing. Based on the Bulk Synchronous Parallel (BSP) model, they execute user-defined operations at vertices iteratively and exchange information between vertices by messages. Even though the native BSP systems (e.g. Pregel and Giraph) execute the operations on vertices synchronously within an iteration, many other platforms (e.g. Grace, Blogel and GraphHP) have proposed asynchronous execution for improved efficiency. However, they also bring about an undesirable side effect: a program designed for synchronous platforms may not run properly on asynchronous platforms. In this demo, we present DFA-G (*D*eterministic *F*inite *A*utomaton for *G*raph processing), a unified programming model for vertex-centric parallel graph processing. Built on DFA, DFA-G expresses the computation at a vertex as a process of message-driven state transition. A program modeled after DFA-G can run properly on both synchronous and asynchronous platforms. We demonstrate how to build DFA-G models using a graphical user interface and also how to automatically translate them into BSP programs.

2 INTRODUCTION

A wide variety of applications (e.g. social networks and biological networks) use graphs to represent the entities and the interactions between them. Due to the massive scale of these graphs, graph analytics usually have to be performed in parallel over a cluster of machines. As a result, a lot of systems based on the classical Bulk Synchronous Parallel (BSP) model have been built for this purpose. However, the native BSP implementations (e.g. Pregel [1] and Giraph [2]) may cause substantial inefficiency due to frequent synchronization and communication among computing nodes. Therefore, alternative platforms (e.g. Giraph Unchained [3], Grace [4], Blogel [5], GraphHP [6], Maiter [7] and HAGP [8]) have been proposed to facilitate asynchronous execution on BSP programs for improved efficiency. Unfortunately, they also bring about an undesirable side effect: a BSP program designed for synchronous platforms may not run properly on asynchronous platforms.

We illustrate program incompatibility between synchronous and asynchronous platforms by the example of bipartite matching. The problem of bipartite matching is to find a maximal matching, in which no additional edge can be added without sharing an end point, in a bipartite graph. A BSP program on synchronous platforms [1] implements a four-stage handshake. In the 1st stage, each unmatched left vertex sends *Request* messages to its neighbors to request match. In the 2nd stage, each unmatched right vertex randomly chooses one of the *Request* messages it receives, sends a corresponding *Grant* message, and sends *Deny* messages to other requesters. In the 3rd stage, each unmatched left vertex chooses one of the grants it receives and sends a corresponding *Accept* message. In the 4th stage, an unmatched right vertex receives the *Accept* message and records its corresponding left vertex. Unfortunately, we observe that the program written for synchronous platforms can not be directly reused on asynchronous ones in this example. With asynchronous execution, a right vertex may simultaneously have a *Request* and an *Accept* message in its message queue. The synchronous program however could not properly handle a hybrid message queue.

Since a BSP program written for synchronous platforms may not work properly on asynchronous ones, end users may thus be required to design different parallel algorithms for different platforms. Therefore, there is a need for a unified programming model such that a BSP program written according to the model can be guaranteed to work properly on both synchronous and asynchronous platforms. To this aim, we present this demo. Our major contributions can be summarized as follows:

- We propose a unified programming model, DFA-G, for vertex-centric parallel graph processing.
- We demonstrate how to construct DFA-G models using a graphical user interface.
- We demonstrate how to automatically translate DFA-G models into runnable BSP programs.

3 System Overview

3.1 BSP Programming Interface

The BSP data model is a directed graph in which each vertex is uniquely identified by an identifier. The synchronous BSP computation consists of a sequence of supersteps. During a superstep, it invokes a uniform, user-defined function at each vertex, conceptually in parallel. The function specifies the behavior at a vertex and a single superstep. To enable complex computations, BSP systems usually allows users to define parameters on vertices and edges,

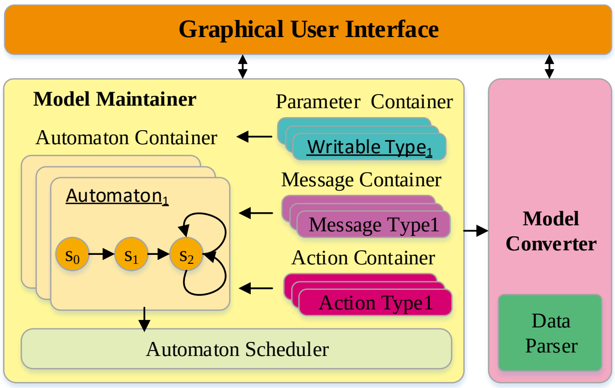


Figure 3.1: System Architecture

whose values can be updated by vertex function. Programming on BSP platforms primarily involves defining the function for graph vertices.

On the open-source Giraph platform, users can instruct the behaviors of vertices in the primary `Vertex` class. It has the `Compute()` method, which specifies the actions taken at vertices. `Compute()` can inspect the received messages via a message iterator and send messages using the method `sendMessage()`. It can also query and update the state of a vertex using the methods `getValue()` and `setValue()` respectively.

On asynchronous platforms, the behaviors of vertices are usually specified in the conceptually same way. Without the synchronization barrier between supersteps, the computations on vertices can however be executed in any order as specified by default setting or end users.

3.2 System Architecture

The architecture, as shown in Figure 3.1, consists of three components: Graphical User Interface (GUI), Model Maintainer (MM) and Model Converter (MC). GUI provides users with a graphical interface, with which DFA-G models can be iteratively constructed by simply clicking and dragging. MM is for managing DFA-G models. It has an automaton container, a message container, an action container and an automaton scheduler. Automaton container records the automatons. Message and action containers record the details of the messages and actions used in automatons respectively. Automaton scheduler is responsible for arranging the execution order of automatons. Finally, MC is responsible for translating a DFA-G model into a runnable BSP program.

4 The Unified Programming Model

4.1 DFA-G Model

In automata theory, DFA is formally represented by a 5-tuple, $(Q, \Sigma, \Delta, q_0, F)$, where Q denotes a finite set of states, Σ denotes a finite set of input symbols, Δ denotes a transition function $\Delta : Q \times \Sigma \rightarrow Q$, q_0 denotes an initial state in Q, and F denotes a set of final states where $F \subseteq Q$.

To recognize a sequence of characters over Σ, a DFA starts from state q_0 and then transitions from one state to another according to the given character and its transition function Δ.

We observe that the BSP computation is essentially driven by messages. It motivates us to express the computation at a vertex as a process of message-driven state transition. Formally, DFA-G models vertex computation by a 5-tuple, $(\mathscr{S}, \mathscr{M}, \mathscr{A}, \mathscr{T}, s_0)$, in which:

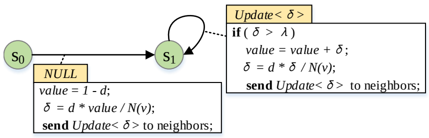
- $\mathscr{S}$ denotes a finite set of states that a vertex can be in.
- $\mathscr{M}$ denotes a finite set of message types that are exchanged between vertices. Message definition may contain updatable parameters. They are usually used to carry the values that must be transferred between vertices.
- $\mathscr{A}$ denotes a finite set of action types that a vertex should take as a result of state transition.
- $\mathscr{T}$ denotes a transition function $\mathscr{T} : \mathscr{S} \times \mathscr{M} \xrightarrow{\mathscr{A}} \mathscr{S}$. The function specifies the state transition at a vertex upon receiving a message and the action it needs to take.
- s_0 denotes an initial state of vertices at the beginning of computation. Note that initially, no message exists in an automaton. Therefore, in the definition of $\mathscr{T}$, s_0 usually has to make a state transition *unconditionally* without being triggered by any message.

Similar to the native DFA, DFA-G incurs state transition upon receiving a message (except in the initial state s_0). It however assumes that once a vertex completes a state transition, it becomes inactive. An inactive vertex can *only* be reactivated by a new message. Accordingly, in DFA-G, the state transition at a vertex can terminate at any possible state. Therefore, the model does not define the set of final states. Action definition ($\mathscr{A}$) usually involves sending messages to one or more destination vertices and updating the values of vertex, edge and message parameters. The values of these parameters however can not affect the progress of state transition, which is solely determined by the current state of a vertex and the type of the message it receives as defined in $\mathscr{T}$.

By expressing vertex computation as a series of message-driven state transition, DFA-G processes messages in a one-at-a-time manner without regard to their arrival order. Its algorithmic correctness is thus independent of the processing order of messages.

4.2 Case Study

4.2.1 PageRank [9]



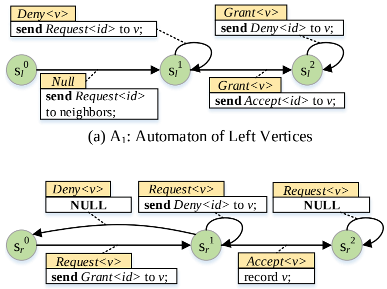
```
value = 1 - d;
 δ = d * value / N(v);
 send Update< δ > to neighbors;
```

```
if ( δ > λ )
   value = value + δ;
   δ = d * δ / N(v);
   send Update< δ > to neighbors;
```

Figure 4.1: Automaton of PageRank

The PageRank values of the vertices in a graph can be computed by the DFA-G model as shown in Figure 4.1, in which *value* represents the PageRank value of a vertex and $N(v)$ represents its number of outgoing neighboring vertices. It instructs the iterative computation process by accumulative updates. Initially, every vertex sets its PageRank value to (1-d), in which d is a damping factor, sends *Update* messages to its neighbors indicating its value update, and then proceeds to state s_1. At state s_1, upon receiving an *Update*$< \delta >$ message, a vertex would update its value by δ. If the value of δ exceeds a predefined threshold of λ, the vertex continues to send out *Update* messages to its neighbors. It can be observed that if the value of λ are set to be sufficiently small, the computed PageRank values would converge to their true values.

4.2.2 Bipartite Matching



(a) A_1: Automaton of Left Vertices

(b) A_2: Automaton of Right Vertices

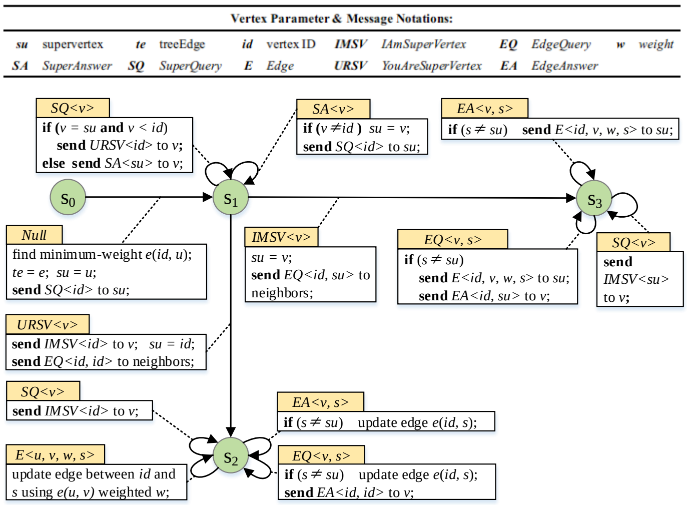
Figure 4.2: Automatons of Bipartite Matching

The DFA-G model for bipartite matching is shown in Figure 4.2, in which the automatons of left and right vertices are presented by A_1 and A_2 respectively and id denotes the identifier of the current vertex. The automatons instruct a three-stage handshake process. At initialization, each left vertex sends a *Request* message to its neighboring right vertices, and then proceeds to the state s_l^1 waiting for the responses from right vertices. In s_l^1, if it receives a *Grant* message from a right vertex, it proceeds to the state s_l^2 and sends an *Accept* message to the grantor. Otherwise, if it receives a *Deny* message from a right vertex, which means the right vertex is engaged but not matched, it continues to send a *Request* message to the right vertex. In s_l^2, if a matched left vertex continues to receives a *Grant* message, it would send a *Deny* message to notify the grantor such that the grantor could accept new matching requests.

In A_2, if a right vertex at s_r^0 receives a *Request* message from a left vertex, it would proceed to s_r^1 and send a *Grant* message to the requestor. In s_r^1, if it receives an *Accept* message from a left vertex, it would proceed to the matched state of s_r^2 and record its matching left vertex. Otherwise, if it continues to receive *Request* messages at s_r^1, it would send *Deny* messages to the requestors to notify them of its engaged state. If it receives a *Deny* message at s_r^1, it would transit back to the state of s_r^0, which indicates that it is free to accept a new matching request.

4.2.3 Minimum Spanning Tree

The general idea of the MST algorithm is to iteratively find conjoined-trees in a connected graph [10]. A conjoined-tree is a directed graph consisting of the minimum-weight edges of vertices and has two cycled vertices at its root. The algorithm first find all the conjoined-trees in a graph and then fold each of them into a single vertex. The process is repeated until the entire graph becomes a single vertex. The minimum spanning tree consists of the edges in the resulting conjoined-trees.



Vertex Parameter & Message Notations:
su supervertex
te treeEdge
id vertex ID
IMSV IAmSuperVertex
EQ EdgeQuery
w weight
SA SuperAnswer
SQ SuperQuery
E Edge
URSV YouAreSuperVertex
EA EdgeAnswer
S_0
S_1
S_2
S_3
SQ<v>
if (v = su and v < id) send URSV<id> to v; else send SA<su> to v;
SA<v>
if (v ≠ id) su = v; send SQ<id> to su;
EA<v, s>
if (s ≠ su) send E<id, v, w, s> to su;
Null
find minimum-weight e(id, u); te = e; su = u; send SQ<id> to su;
IMSV<v>
su = v; send EQ<id, su> to neighbors;
EQ<v, s>
if (s ≠ su) send E<id, v, w, s> to su; send EA<id, su> to v;
SQ<v>
send IMSV<su> to v;
URSV<v>
send IMSV<id> to v; su = id; send EQ<id, id> to neighbors;
SQ<v>
send IMSV<id> to v;
EA<v, s>
if (s ≠ su) update edge e(id, s);
E<u, v, w, s>
update edge between id and s using e(u, v) weighted w;
EQ<v, s>
if (s ≠ su) update edge e(id, s); send EA<id, id> to v;

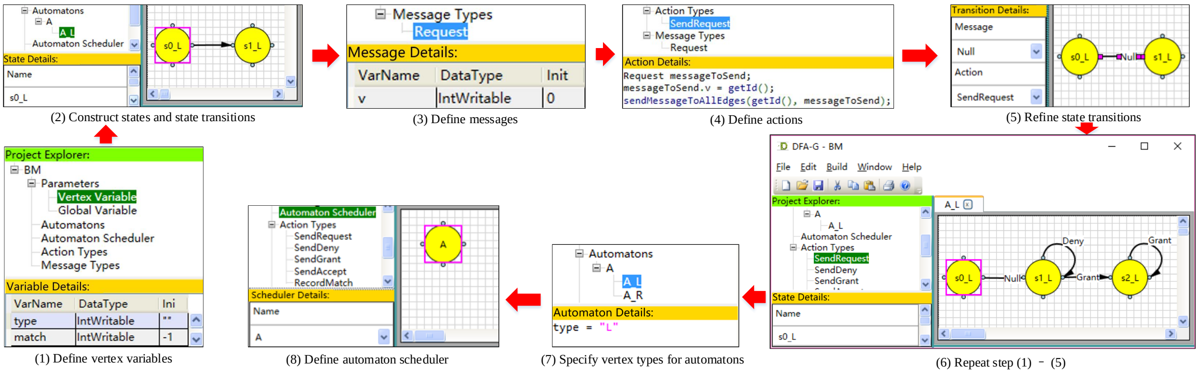
Figure 4.3: Automaton of Minimum Spanning Tree

The automaton of the MST algorithm is shown in Figure. 4.3. It instructs the process of constructing and folding conjoined-trees. In the demo system, the DFA-G model uses automaton scheduler to instruct iterative execution of the automaton. The automaton selects the one with smaller ID of two cycled root vertices in a conjoined-tree as *supervertex*. Each supervertex corresponds to a conjoined-tree. The messages of *SuperQuery*, *SuperAnswer*, *YouAreSupervertex* and *IAmSupervertex* are for supervertex identification. The messages of *EdgeQuery*, *EdgeAnswer* and *Edge* are for graph folding. More explanations on the automaton are omitted here due to space limit.

5 Evaluation and Demonstration

We have implemented a demo system [11] to generate BSP programs for the Giraph programming interface by constructing DFA-G models.

Model Construction. Automaton construction primarily consists of the following steps: (1) define vertex variables; (2) create states and state transitions; (3) define messages and actions; (4) refine state transitions with the defined messages and actions. Some algorithms (e.g. BM) may require different types of vertices to execute different automatons. The system allows users to specify the type of vertices for an automaton. For instance, in the BM example, as shown in 5.1, A_L and A_R are the automatons modeled for left and right vertices respectively. On



(2) Construct states and state transitions

VarName	DataType	Init
v	IntWritable	0

(3) Define messages

```
Request messageToSend;
messageToSend.v = getId();
sendMessageToAllEdges(getId(), messageToSend);
```

(4) Define actions

(5) Refine state transitions

VarName	DataType	Ini
type	IntWritable	""
match	IntWritable	-1

(1) Define vertex variables

(8) Define automaton scheduler

```
type = "L"
```

(7) Specify vertex types for automatons

(6) Repeat step (1) – (5)

Figure 5.1: Model Construction Workflow

the interface for automaton scheduler, users can create directed edges between automatons to control their execution order. The system allows users to define an automaton-transition condition on each edge. If there is a cycle (e.g. in the case of MST), an automaton-transition condition on one of cycled edges is necessary for the termination for model execution. All the tasks primarily involve simply clicking and dragging on the interface. An example of model construction for BM is demonstrate in Figure 5.1.

Model Conversion. The process of model conversion starts with generating the vertex and message classes. The `Compute()` function of the vertex class is outlined as two-level nested if-statements: the first-level if-condition is specified on vertex state and the second-level one is on message type. Since a vertex becomes inactive after it completes a state transition, a `voteToHalt()` operation is always inserted at the end of the `Compute()` function. If different types of vertices are supposed to execute different types of automatons, the corresponding `Compute()` function has an additional level of nested if-statements, whose if-condition checks the vertex type. The function of automaton scheduler is implemented in the `main()` function. Multiple automatons are connected by a series of if-statements, whose if-conditions correspond to user-specified automaton-transition conditions. The repeated invocation of an automation is similarly implemented by a while-statement.

Performance Evaluation. We demonstrate the performance of the programs modeled after DFA-G on Giraph, Grace and GraphHP. On Giraph, we also compare the performance of the DFA-G programs with that of corresponding synchronous programs. We have implemented the algorithms for PageRank and BM, which are run on the open-source Web-Google and Hamrle3 datasets [12] respectively.

The programs are run on a machine that has the memory size of 16G, disk storage of 500G and 16 AMD Opteron(TM) processors of 2.6GHz frequency. The running system has 10 parallel computing nodes. The performance of different programs are presented in Table. 5.1. It can be observed that the asynchronous platforms can achieve considerably better performance than synchronous ones. On Giraph, the asynchronous DFA-G program (Giraph(asyn)) of BM achieves similar performance to the synchronous one (Giraph(syn)). The asynchronous PageRank produces more messages than the synchronous program and converges faster on the asynchronous platforms.

Demonstration Plan. The system will be demonstrated on a personal computer. The at-

Table 5.1: Performance Demonstration

Time(s)	Giraph(syn)	Giraph(asyn)	GraphHP	Grace
BM	72.32	80.20	39.10	5.21
PageRank	69.54	118.24	22.10	15.58

tendees will be able to design a wide range of graph algorithms and execute the resulting programs on Giraph, GraphHP and Grace. They will also be able to observe the performance comparisons of the programs designed for synchronous platforms and the asynchronous programs modeled after DFA-G.

6 Conclusion

In this demo, we propose a unified programming model, DFA-G, for vertex-centric parallel graph processing. DFA-G ensures that a BSP program modeled after it can run properly on both synchronous and asynchronous platforms. We have demonstrated how to construct a DFA-G model and automatically transform a model into a runnable BSP program. Our demo is currently built for the Giraph programming interface. But it can easily generalize to other BSP programming interfaces.

References

[1] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD*, 2010, pp. 135–146.

[2] C. Avery, "Giraph: Large-scale graph processing infrastructure on hadoop," *Proceedings of the Hadoop Summit, Santa Clara*, 2011.

[3] M. Han and K. Daudjee, "Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 9, pp. 950–961, 2015.

[4] G. Wang, W. Xie, A. J. Demers, and J. Gehrke, "Asynchronous large-scale graph processing made easy," in *CIDR*, 2013.

[5] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Blogel: A block-centric framework for distributed computation on real-world graphs," *Proceedings of the VLDB Endowment*, vol. 7, no. 14, pp. 1981–1992, 2014.

[6] Q. Chen, S. Bai, Z. Li, Z. Gou, B. Suo, and W. Pan, "GraphHP: A hybrid platform for iterative graph processing," Northwestern Polytechnical University, Tech. Rep., 2014. [Online]. Available: http://www.wowbigdata.cn/paper/GraphHP\%EF\%BC\%9AA\%20Hybrid\20Platform\%20for\%20Iterative\%20Graph\%20Processing.pdf

[7] Y. Zhang, Q. Gao, L. Gao, and C. Wang, “Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 8, pp. 2091–2100, 2014.

[8] T. Gao, Y. Lu, and B. Zhang, “HAGP: A hub-centric asynchronous graph processing framework for scale-free graph,” in *The 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2015, pp. 789–792.

[9] S. Brin and L. Page, “The anatomy of a large-scale hypertextual web search engine,” *Computer networks*, vol. 56, no. 18, pp. 3825–3833, 2012.

[10] S. Chung and A. Condon, “Parallel implementation of Bouvka’s minimum spanning tree algorithm,” in *Proceedings of IPPS*, 1996, pp. 302–308.

[11] “The DFA-G demo system, Northwestern Polytechnical University,,” http://www.wowbigdata.cn/dfa-g/demo.html.

[12] “The University of Florida sparse matrix collection,” http://www.cise.ufl.edu/research/sparse/matrices/.