GraphU: A Unified Vertex-Centric Parallel Graph Processing Platform

Jing Su, Qun Chen, Zhuo Wang, Murtadha Ahmed, Zhanhuai Li
School of Computer Science, Northwestern Polytechnical University, Xi'an, ShaanXi, China
jinjin-su@163.com,{chenbenben@, wzhuo918@mail., a.murtadha@mail.,lizhh@}nwpu.edu.cn

***Abstract*—Many synchronous and asynchronous distributed platforms based on the Bulk Synchronous Parallel (BSP) model have been built for large-scale vertex-centric graph processing. Unfortunately, a program designed for a synchronous platform may not work properly on an asynchronous one. As a result, given the same problem, end users may be required to design different parallel algorithms for different platforms. Recently, we have proposed a unified programming model, DFA-G (Deterministic Finite Automaton for Graph processing), which expresses the computation at a vertex as a series of message-driven state transitions. It has the attractive property that any program modeled after it can run properly across synchronous and asynchronous platforms.**

In this demo, we first propose a framework of complexity analysis for DFA-G automaton and show that it can significantly facilitate complexity analysis on asynchronous programs. Due to the existing BSP platforms' deficiency in supporting efficient DFA-G execution, we then develop a new prototype platform, GraphU. GraphU was built on the popular open-source Giraph project. But it entirely removes synchronization barriers and decouples remote communication from vertex computation. Finally, we empirically evaluate the performance of various DFA-G programs on GraphU by a comparative study. Our experiments validate the efficacy of the proposed complexity analysis approach and the efficiency of GraphU.

Video: http://www.wowbigdata.com.cn/GraphU/video.html.

I. Introduction

A lot of systems based on the BSP model have been built for large-scale parallel graph processing in a distributed environment. However, the native BSP implementations (e.g. Pregel [1] and Giraph [2]) may cause substantial inefficiency due to frequent synchronization and communication among parallel workers. Therefore, many alternative platforms (e.g. Giraph Unchained [3] and GraphHP [4]) have been proposed to facilitate asynchronous execution on BSP programs for improved efficiency. Unfortunately, asynchronism also brings about an undesirable side effect: a BSP program designed for synchronous platforms may not run properly on asynchronous ones. As a result, given the same problem, end users usually have to design different parallel algorithms for different platforms. To solve this incompatibility, we have recently proposed a unified programming model, DFA-G [5]. Expressing the computations at a vertex as a series of message-driven state transitions, DFA-G can ensure that a program modeled after it can work properly across synchronous and asynchronous platforms.

Even though complexity analysis on *synchronous* BSP programs has been extensively studied in the literature [6], complexity analysis on *asynchronous ones* is usually much more complicated, thus still remains an open challenge. DFA-G has the attractive property that vertex computations are *entirely* driven by received messages. Accordingly, the computational complexity of a DFA-G automaton coincides with the total number of messages exchanged between vertices. Therefore, complexity analysis on an asynchronous DFA-G program can be greatly simplified by performing complexity analysis on its corresponding automaton.

On the other hand, even though the existing asynchronous platforms can to some extent alleviate the inefficiency resulting from frequent synchronizations among workers, they did not entirely remove the requirement of global synchronization. Moreover, they can only optimize vertex execution order by simple default or user-specified settings. The central concepts of DFA-G, state and state transition, were not even taken into their design consideration. As a result, they can not effectively support efficient execution of DFA-G programs. To this end, we develop a new prototype platform, GraphU, based on the popular open-source Giraph project [2]. The major contributions of this demo can be summarized as

1) We propose a framework of complexity analysis for DFA-G automaton and show that it can significantly simplify complexity analysis on asynchronous BSP programs; (**Section 2**)
2) We develop a new prototype system, GraphU, which can effectively optimize the execution efficiency of DFA-G programs; (**Section 3**)
3) We empirically evaluate the performance of various DFA-G programs on GraphU. Our experimental results validate the efficacy of the proposed complexity analysis approach and the efficiency of GraphU. (**Section 4**)

II. DFA-G Programming Model

A. Model Overview

Formally, a DFA-G automaton models vertex computation by a 5-tuple, $(\mathcal{S}, \mathcal{M}, \mathcal{A}, \mathcal{T}, \mathcal{S}_{\mathcal{O}})$, in which:

- $\mathcal{S}$ denotes a finite set of vertex states.
- $\mathcal{M}$ denotes a finite set of types of the messages exchanged between vertices.

- $\mathcal{A}$ denotes a finite set of types of the actions taken by a vertex upon receiving a message.
- $\mathcal{T}$ denotes a transition function $\mathcal{T} : \mathcal{S} \times \mathcal{M} \xrightarrow{\mathcal{A}} \mathcal{S}$. The function specifies the state transition at a vertex upon receiving a message and the action it needs to take.
- S_O denotes an initial state of vertices. Initially, no message exists in an automaton. Therefore, in the definition of $\mathcal{T}$, $\mathcal{S}_\mathcal{O}$ 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_O). 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. Since state transition can terminate at any possible state, the automaton does not need to specify final states. In the automaton definition, a message definition ($\mathcal{M}$) can contain updatable parameters, which are usually used to transfer the values between vertices. An action definition ($\mathcal{A}$) usually involves sending messages to one or more destination vertices and updating the values of vertex, edge and message parameters. In DFA-G, state transition at a vertex is solely determined by its current state and the type of the message it receives. The values of the updatable parameters can not affect the progress of state transition. *By expressing vertex computation as a series of message-driven state transitions, 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.*

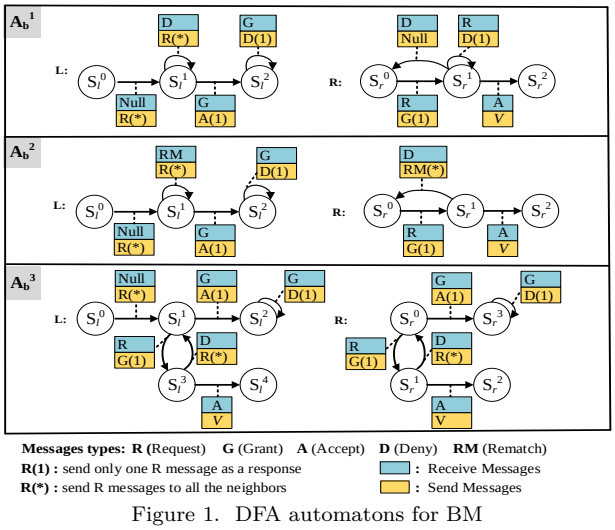


Figure 1. DFA automatons for BM

Three different DFA-G automatons for the problem of bipartite matching (BM) are shown in Figure 1, in which the automatons of left and right vertices are presented separately. Given a bipartite graph, the BM problem is to find a maximal matching, in which no additional edge can be added without sharing an end point. Detailed explanations on the presented automatons are omitted here due to space constraint, but can be found in our technical report [7].

B. Framework of Complexity Analysis for Automaton

The computational cost of a DFA-G program includes both the cost of DFA-G automaton (state transition) and the cost of specified vertex computations. In this subsection, we define the soundness and complexity of a DFA-G automaton. *Note that the computational complexity of a DFA program can be easily estimated based on the complexity results of its corresponding automaton and vertex computations.*

Automaton Soundness. Note that the total number of messages generated by an automaton running on a graph depends on the graph and the order of vertex execution as well as the automaton itself. Soundness analysis considers the worst case of running an automaton on a graph. Formally, we define the soundness of a DFA-G automaton as follows:

Definition 1: A DFA-G automaton, A, is sound *if and only if* $\forall$ *G and* $\forall$ χ*, in which G represents a graph and* χ *represents an instance of vertex execution order, the number of messages generated by running A on G by the order of* χ *is finite.*

Intuitively, if A is *not* sound, then there exists a graph and an instance of vertex execution order s.t. A can not even terminate on the graph. For instance, the automaton presented in Figure. 1 is not sound. Consider the following vertex execution sequence: a right vertex (v_i) in the state of S_r^1, upon receiving a *request* message from a left vertex (v_j) in the state of S_l^1, sends a *deny* message to v_j, v_j in return sends a *request* message to v_i. The actions of v_i and v_j result in a message loop. The automaton therefore can not terminate in the worst case.

Automaton Complexity. If a DFA-G automaton is sound, we measure its computational complexity by its worst-case running cost, which corresponds to the total number of messages generated in the worst case. Formally, the computational complexity of an automaton is defined as follows:

Definition 2: A sound DFA-G automaton, A, has the computational complexity of $\boldsymbol{O}(\omega)$ *if and only if* $\forall$ *G and* $\forall$ χ*, the number of messages generated by running A on G by the order of* χ *is bounded by* $\boldsymbol{O}(\omega)$*, in which G represents a graph and* χ *represents an instance of vertex execution order.*

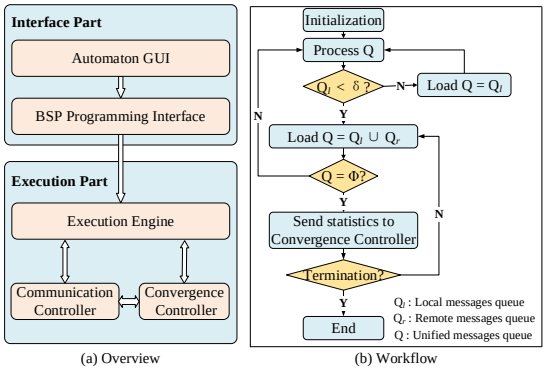
Consider the automaton for BM, A_b^2, shown in Figure. 1. Firstly, the automaton is sound because it contains no message loop. Secondly, a left vertex can deny its neighboring right vertex at most once, it can thus receive at most $\mathbf{O}(K^2)$ *rematch* messages from all its neighboring right vertices, where K denotes the maximal vertex degree in the graph. Therefore, the complexity of the automaton A_b^2 can be represented by $\mathbf{O}(N \cdot K^3)$, in which N denotes the total number of vertices in the graph. The automaton of A_b^3

as shown in Figure. 1 however has the better complexity of $\mathbf{O}(N \cdot K^2)$. It reduces the complexity by allowing both left and right vertices to initiate the handshake process.

*It is worthy to point out that the complexity of an automaton may not necessarily coincide with the parallel efficiency of its corresponding program. Besides automaton complexity, parallel efficiency also depends on communication latency and workload balance. Nonetheless, combined with the complexity of specified vertex computations, a big-***O** *automaton complexity result can provide with an upperbound estimate on the efficiency of a DFA-G program. A a result, it can serve as an effective performance indicator of its corresponding parallel program.*

Automaton Execution Optimization. Soundness and complexity analysis estimate the worst-case computational cost of an automaton. However, the actual running cost of an automaton may to a large extent depend on vertex execution order. The influence of vertex execution order on program efficiency has also been widely recognized in the empirical study conducted on existing asynchronous systems [8]. The existing systems usually optimize vertex execution order based on vertex parameter values or message list size. In addition to these metrics, the DFA-G automaton provides with a more intuitive (in terms of user understanding) and more effective (in terms of performance improvement) option based on *vertex state*. For instance, consider the automaton, A_b^1, shown in Figure. 1. In the worst case, it can not even terminate. However, if a left vertex in the state of S_l^2 is always processed before a right vertex in the state of S_r^1 if they are simultaneously active, the message loop would be broken. Its efficiency can instead be bounded by $\mathbf{O}(N \cdot K^3)$. For more examples on state-based vertex execution optimization, please refer to our technical report [7].

III. GraphU System



(a) Overview (b) Workflow

Figure 2. GraphU Platform

We have developed a new prototype system, GraphU, to effectively support efficient execution of DFA-G programs. GraphU was built on the open-source Giraph project[2]. The platform overview of GraphU is presented in Figure. 2 (a). It consists of two parts: **interface** and **execution**. GraphU provides with a GUI interface, in which users can specify a DFA-G automaton by simple clicking and drawing. It transforms an automaton into a runnable BSP code. GraphU shares the same code interface with Giraph. The execution part contains three components: the execution engine, the communication controller and the convergence controller. The execution engine schedules and executes vertex computations, the communication controller is responsible for communication between workers, and the convergence controller automatically detects the termination of a running automaton. In the rest of this section, we briefly describe our implementation of execution engine. More details on the system implementations can be found in the technical report [7]. All the source codes can also be downloaded at [7].

The workflow of the execution engine at a given worker is presented in Figure. 2 (b). Since running a DFA-G automaton does not require any global synchronization, GraphU entirely removes global synchronization and decouples vertex computations from remote communication. Each worker maintains two message queues, one for the messages destined for the local worker and the other for the messages destined for remote workers. After initialization, it would retrieve the messages from both queues into main memory and organize them by their destination vertices in a **hashmap**. Vertex computations are then sequentially executed until the message hashmap becomes empty. Vertex computation usually triggers state transitions and meanwhile generates new messages. The new messages are sent to their respective queues according to their destinations. In a distributed environment, the communication between workers is usually much more prohibitive than the communication between vertices within the same worker. Therefore, in GraphU, a worker would first process the local messages. It would retrieve the messages sent from remote workers if and only if the number of local messages to be processed falls below a pre-specified threshold. If execution priority is specified on vertices, each worker would first order the active vertices by their priority degrees and then sequentially process the vertices. In our current implementation, priority can be set based on vertex state, vertex value or the number of messages. More options can however be similarly implemented on GiraphU.

IV. Evaluation and Demo Plan

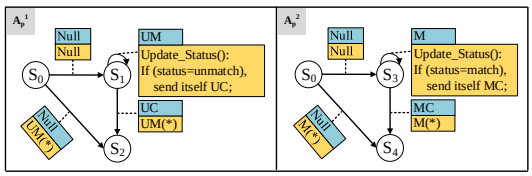


Figure 3. DFA automatons for PM

We empirically evaluate the performance of different DFA-G programs for Bipartite Matching (BM) and Pattern Matching (PM) on real graphs. The goal of PM is to find all the matches of a given graph pattern. Only preserving the child relationships of each vertex, PM provides a

Table I
PERFORMANCE COMPARISON OF DFA-G PROGRAMS

	BM			PM	
	A_b^1	A_b^2	A_b^3	A_p^1	A_p^2
No. of Messages (mil)	fail	135.83	**102.89**	9.65	**4.65**
Runtime (s)	fail	46.08	**31.90**	34.09	**24.42**

practical alternative to subgraph isomorphism by relaxing its stringent matching conditions [9]. We have implemented the three DFA-G automatons shown in Figure. 1 for BM. For PM, we have implemented the two DFA-G automatons shown in Figure. 3. It can be observed that both automatons of A_p^1 and A_p^2 have the complexity of $\mathbf{O}(|V|+|E|)$, in which $|V|$ and $|E|$ denote the numbers of vertices and edges in a graph respectively. In A_p^1, a vertex would propagate its status to its parents if and only if its status changes from *match* to *unmatch*. In contrast, in A_p^2, a vertex would propagate its status to its parents if and only if its status changes from *unmatch* to *match*. We also compare GraphU with the alternative platforms sharing the same BSP programming interface, Giraph and GiraphUC, which are synchronous and asynchronous respectively. Note that even though GiraphUC is asynchronous, it still requires global synchronization.

The BM programs are run on the real dataset of com-orkut [1] and the PM programs are run on the real dataset of ACM-citation [2]. All the programs are run on a cluster consisting of 7 machines, each of which is equipped with a memory of 16 G, a disk storage of 500G and 16 AMD Opteron processors of 2.6GHz frequency. We compare the performance of DFA-G programs on the metrics of the number of generated messages and the consumed runtime. Due to DFA-G's execution uncertainty, we report the results averaged over three runs. On GraphU, all the programs were run without priority setting. Due to space limit, please refer to our technical report [7] for the effect of priority setting on DFA-G's execution performance.

The performance comparison between different DFA-G programs is presented in Table. I. On BM, it can be observed that A_b^3 performs better than A_b^2, which in turn performs better than A_b^1. A_b^1 in all the three runs generates so many messages s.t. the system collapses due to memory overflow. These experimental results are consistent with the theoretical complexity analysis results on DFA-G automatons. The experimental results on PM are similar. A_p^2 performs considerably better than A_p^1 on both metrics. In the real test graph, the number of matching (including partially matching) vertices is much less than the number of unmatching vertices. These experimental results demonstrate that automaton complexity analysis can effectively predict the parallel performance of DFA-G programs.

The experimental results of comparing GraphU with Giraph and GiraphUC are also presented in Table. II. All the platforms run the same DFA-G programs. For BM and PM, we run the DFA-programs of A_b^3 and A_p^2 respectively. It can be observed that GraphU performs considerably better than both Giraph and GiraphUC on runtime. GraphU can significantly improve performance by entirely removing global synchronization and decoupling vertex computation from remote communication. These observations validate the efficiency of GraphU.

[1]http://snap.stanford.edu/data/index.html#communities
[2]https://www.aminer.cn/citation

Table II
GRAPHU VS GIRAPH AND GIRAPHUC

runtime (s)	Giraph	GiraphUC	GraphU
A_b^3	53.51	62.63	**31.90**
A_p^2	42.547	37.30	**24.42**

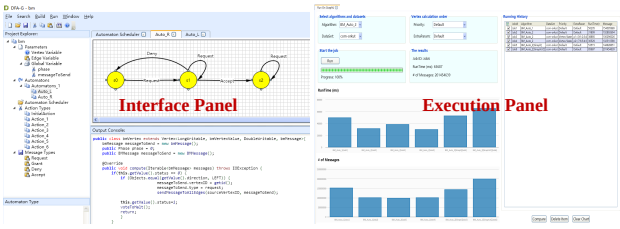


Figure 4. The demo system of GraphU

Demo Plan. The demo system of GraphU, whose screenshots are presented in Figure. 4, consists of two panels, interface panel and execution panel. In the interface panel, users can manually construct a DFA-G automaton by simple clicking and drawing operations, and the system would automatically translate a constructed automaton into an executable BSP code. In the execution panel, users can run different DFA-G programs on different platforms and compare their performance. The attendees will be invited to construct automatons and run DFA-G programs using different datasets on a laptop.

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 *Proceedings of the ACM International Conference on Management of data (SIGMOD)*, 2010, pp. 135–146.

[2] "Apache giraph." [Online]. Available: http://giraph.apache.org/

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

[4] Q. Chen, S. Bai, Z. Li, Z. Gou, B. Suo, and W. Pan, "Graphhp: A hybrid platform for iterative graph processing," 2017. [Online]. Available: https://arxiv.org/pdf/1706.07221.pdf

[5] B. Suo, J. Su, Q. Chen, Z. Li, and W. Pan, "Dfa-g: A unified programming model for vertex-centric parallel graph processing," in *IEEE 16th International Conference on Data Mining, Demo Track*, 2016, pp. 1328–1331.

[6] L. Qin, J. X. Yu, L. Chang, H. Cheng, C. Zhang, and X. Lin, "Scalable big graph processing in mapreduce," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 827–838.

[7] GraphU. [Online]. Available: http://www.wowbigdata.com.cn/GraphU/demo.html

[8] G. Wang, W. Xie, A. J. Demers, and J. Gehrke, "Asynchronous large-scale graph processing made easy." in *CIDR*, vol. 13, 2013, pp. 3–6.

[9] A. Fard, M. U. Nisar, L. Ramaswamy, J. A. Miller, and M. Saltz, "A distributed vertex-centric approach for pattern matching in massive graphs," in *IEEE International Conference on Big Data*. IEEE, 2013, pp. 403–411.