SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

GraphU: A Unified Vertex-Centric Parallel Graph Processing Platform

Jing Su, Qun Chen, Zhuo Wang, Murtadha Ahmed and Zhanhuai Li

2018-01-16

1 ABSTRACT

Many synchronous and asynchronous distributed platforms have been built for large-scale vertex-centric graph processing. Built on the Bulk Synchronous Parallel (BSP) model, the synchronous platforms iteratively perform vertex computations in a strict batch mode. In contrast, the asynchronous platforms enable more flexible vertex execution order and can usually result in better efficiency. Unfortunately, asynchronism also brings about an undesirable side effect: a program designed for a synchronous platform may not work properly on an asynchronous one. As a result, 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 BSP program modeled after it can run properly across synchronous and asynchronous platforms.

In this reprot, we first propose a complexity analysis framework for DFA-G automaton, which can significantly simplify complexity analysis on asynchronous programs. Due to the existing BSP platforms' deficiency in supporting efficient DFA-G execution, we then develop a new propotype 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 conduct an empirical study to evaluate the performance of various DFA-G programs on GraphU and the effect of different priorities on the performance of programs. We also compare them with their alternatives on the existing BSP platforms. Our experiments validate the efficacy of the proposed automaton complexity analysis techniques and the efficiency of GraphU as well as the efficiency of vertex execution sequence .

2 INTRODUCTION

A lot of systems based on the classical Bulk Synchronous Parallel (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. Therefore, 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 BSP 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 programs* is usually much more complicated, thus still remains an open challenge. One important property of DFA-G is that vertex computations are *entirely* driven by received messages. As a result, the computational complexity of a DFA-G automaton coincides with the total number of messages exchanged between vertices. Complexity analysis on an asynchronous program can be greatly simplified by estimating the complexity of its corresponding DFA-G automaton.

The existing asynchronous BSP platforms can to some extent alleviate the inefficiency resulting from frequent global synchronizations among workers. However, 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 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. GraphU was built based on the popular open-source Giraph project [2]. But it entirely removes global synchronization barriers and decouples vertex computation from remote communication. The major contributions of this demo can be summarized as

1. We propose a framework for DFA-G automaton complexity analysis, which can significantly simplify complexity analysis on asynchronous BSP programs; (**Section 3**)

2. We develop a new prototype system, GraphU, which can effectively optimize the execution efficiency of DFA-G programs; (**Section 4**)

3. We empirically evaluate the performance of various DFA-G programs. Our experimental results validate the efficacy of complexity analysis based on DFA-G automaton and the efficiency of GraphU; (**Section 7**)

4. We further optimize the DFA-G programs with different priority scheduling strategy and illustrate the impact of priority on performance by experiments .(**Section 5 and Section 7**)

3 DFA-G PROGRAMMING MODEL

3.1 MODEL OVERVIEW

Formally, a DFA-G automaton 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 types of the messages, which can be exchanged between vertices;
- $\mathscr{A}$ denotes a finite set of types of the actions, which can be taken by a vertex upon receiving a message;
- $\mathscr{T}$ 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. 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. 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 ($\mathscr{M}$) can contain updatable parameters, which are usually used to transfer the values between vertices. An action definition ($\mathscr{A}$) usually involves sending messages to one or more destination vertices and updating the values of vertex, edge and message parameters. However, the values of these parameters can not affect the progress of state transition. In DFA-G, state transition at a vertex is solely determined by its current state and the type of the message it receives as defined in $\mathscr{T}$. *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.*

Three example DFA-G automatons for bipartite matching (BM) are shown in Figure 3.1, 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. Thus, A_b^1 ,A_b^2 and A_b^3 are representing three different processing logic of BM, respectively . In A_b^1 , the left vertices firstly broadcast requests to all right neighbors , when a right vertex S_r^0 receiving the first request response the sender with grant message and transfer to status1,however,if it continue to receive other requests will response with deny message to inform that it has granted the front vertex and cannot grant other left vertices . After receiving deny message , other unmatched left vertices in status1 continue to send request to all right neighbors seeking suitable partner. Once S_l^1 receiving grant will send accept message marking itself successfully match and transfer to final status2 .

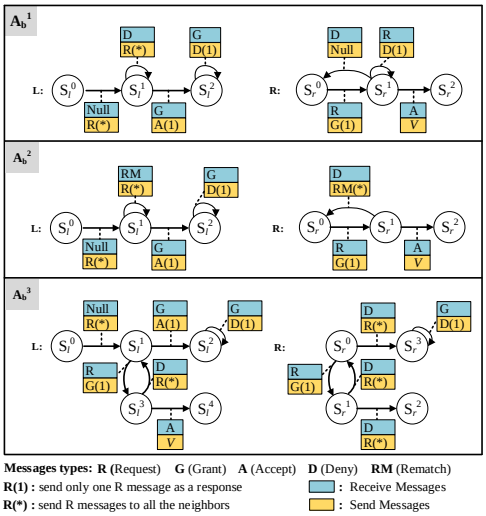

Figure 3.1: DFA automatons for BM

Right vertex S_r^1 receiving accept message transfer to status2 to finish this match.But when left vertex and right vertex both in status1,algorithm may not converge provided S_l^1 receive a deny replayed with a request and the right receiver in S_r^1 will send with deny .

Different from A_b^1 , in order to break message loop in A_b^1 , meanwhile enable to evoke left unmatched vertices can send request to unmatched right vertex , we introduced A_b^2 . A_b^2 added rematch message to stimulate left unmatched vertices to send requests to right vertices that asking for the next round match .Whenever a left vertex, in the state of S_l^2 receives a *grant* message, it would send back a *deny* message to the sender. After receiving the *deny* message, a right vertex v_j in S_r^1 would send a *rematch* messages to all its adjacent left vertices. Upon receiving a *rematch* message, an unmatching left vertex would then initiate a new round of handshake.

To speed up the efficiency of matching each other , we can also let right vertices actively send request , two directions are symmetric ,this is also conform to the common sense . So A_b^3 is a extended version of A_b^1 . Left vertices and right vertices all have rights for sending request and grant message . Once right vertex v_i in S_r^1 receiving deny message will directly send request message instead of passively waiting for left vertices's request. In this way , left vertices and right vertices can simultaneously find their partner to match , algorithm will quickly converge.

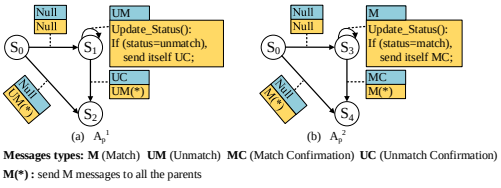

Figure 3.2: DFA automatons for PM

Following is DFA-G automatons for pattern matching(PM) which are shown in Figure 3.2 . Given a query graph $Q(V_q, E_q, l_q)$ and a data graph $G(V, E, l)$, the task is to find all subgraphs of G that match the query Q. The goal of PM[7] is to find all the matches of a given graph pattern . Only preserving the child relationship of each vertex , PM provides a practical alternative to subgraph isomorphism by relaxing its stringent matching conditions .We have implemented the two DFA-G automatons shown in Figure 3.2 . In A_p^1 ,when vertex v_j starting to initialize , it will detect if it‘s label match any vertex‘s label in query graph‘s and if matching then give flag a true value. Supposing that flag is true , then detect whether its children‘s labels also contain corresponding vertex‘s children‘s labels in query graph. In case that , it cannot match query graph‘s vertex will send unmatch message to all itś father nodes . When a vertex v_i receiving a unmatch message ,it will remove this child and re-detect currently whether its children‘s labels also contain corresponding vertex‘s children‘s labels in query graph . As a result , it will send large number of unmatch message in initial step .

In order to reduce messages in initial step, A_p^2 only need vertex that matched send message of match message. In initial step ,the same with A_p^1 ,each vertex detect if its label match query graph‘s label and if matching then give flag a true value . Under the premise that flag is true , it will need to detect whether all its children‘s label contain matched vertex‘s children‘s labels in query graph. Supposing that it can satisfy match condition , it will send to all itś father vertices with match message signifying it has certain to match vertex with same label in query graph . Once a vertex receiving matched message will re-detect this moment if it can match same label vertex in query graph .

3.2 Automaton Complexity Analysis

It can be observed that the computational cost of a DFA-G program includes the cost of DFA-G automaton (state transition) as well as the cost of specified vertex computations. In DFA-G, except in the initialization phase, state transition can only be triggered upon receiving a message. Therefore, the computational cost of a DFA-G automaton increases proportionally with the number of messages exchanged between vertices. 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 actually depends on the graph and the order of vertex execution as well as the automaton itself. Soundness analysis considers all the possibilities 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.*

Definition. 1 specifies the result of the worst-case analysis. 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. 3.1 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 complexity by its worst-case computational running cost, which corresponds to the total number of generated messages in the worst case. Formally, we define the computational complexity of a DFA-G automaton as follows:

Definition 2. *A sound DFA-G automaton, A, has the computational complexity of* $\mathbf{O}(\omega)$ *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 bounded by* $\mathbf{O}(\omega)$*.*

Consider the automaton for BM, A_b^2, shown in Figure. 3.1. Whenever a left vertex, v_i, in the state of S_l^2 receives a *grant* message, it would send back a *deny* message to the grantor, v_j, a right vertex. After receiving the *deny* message, v_j would send a *rematch* messages to all its neighboring left vertices. Upon receiving a *rematch* message, an unmatching left vertex would then initiate a new round of handshake. It can be observed that there is no message loop in the automaton. 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, A_b^3, which is also shown in Figure. 3.1 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 a DFA-G 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 for 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 analyses 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. 3.1. 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 be bounded by $\mathbf{O}(N \cdot K^3)$.

4 GRAPHU SYSTEM

Motivated by the observation that the existing BSP platforms can not effectively support efficient execution of DFA-G programs, we have developed a new prototype system, GraphU. GraphU was built on the open-source Giraph project[2]. Since running DFA-G automaton does not require any global synchronization, GraphU entirely removes global synchronization and decouples vertex computations from remote communication. The computation at each worker, including vertex computation and its communication with other workers, is performed independently without regard to the progress at any other worker.

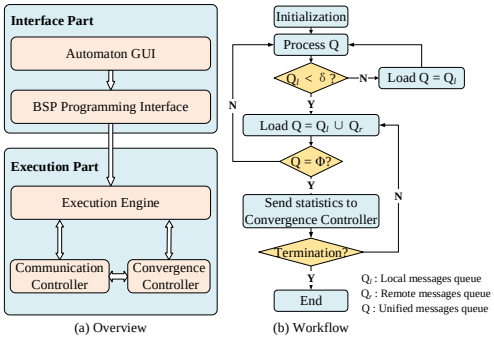

Figure 4.1: GraphU Platform

The platform overview of GraphU is presented in Figure. 4.1 (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 this chapter.All the source codes can also be downloaded at[9].

The workflow of the execution engine at a given worker is presented in Figure. 4.1 (b). 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. It can be observed that in a distributed environment, the communication between workers is usually much more prohibitive than the communication between vertices within the same worker. Frequent worker communications can significantly drag down worker efficiency. 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.

4.1 EXECUTION ENGINE

Execution engine is responsible for computing scheduled vertices with receiving message. Three different scheduling policy will describe in next section .During computation , there are two cocurrentmap ,one is vertexmap storing mapping relation from vertexId to attributes of vertex(eg,status ,priority), and the other is messageStore storing mapping relation from vertexId to messageQueue . When computation , from messageStore we can get vertexList that has receiving message and then together with vertexmap, we can obtain vertexList of higher precedence. In order to quickly acquire vertices with high priority , we use data structure *treemap* to maintain optimal vertex order . The key of *treemap* is priority and value is also a *hashmap* which consists of vertexId and the message belonging to the vertex . Once receiving a message , treemap will be update that it will find the right position new message should insert . Whenever once fetching higher priority vertices to compute , this treemap will update and delete these vertices. We have been implemented *addPartitionMessages* and *getMaxPrioVertices* API on *messageStore*. Before computation , it fetch vertices from *messageStore* , a pairlist consist of vertexID and message . According to specified scheduling policy (or default) to get vertex sets that need to be scheduled. Then it executes the calculation for each vertex through invoking user‘s algorithm until no vertex need to schedule . Then fetch vertexList in remote messageStore to continue computing.

GraphU is completely asynchronous system , two vertices v_i and v_j with different running sequence will produce different number of messages . We first performing local computation using local messageStore that storing messages sending to local worker . When local number of messages are less than δ , following that ,remote messageStore should be used for later computing .

4.2 COMMUNICATION CONTROLLER

In GraphU, remote communication across different workers are completely decoupled from vertex computation. For computation thread and communication thread are independent . During computation , there may be receiving messages from other worker , Since fetching message for computing and adding message are all need to operate the same messagequeue at the same time thus we must add lock on messagequeue . As we all known , lock will make computation inefficient and reduce the concurrency of computing , therefore we can add lock on messagequeue of one vertex that currently computing other than add lock on the all messagequeue. There are two messagequeue for each worker . One is localmessagequeue destined for the local worker , and the other is remotemessagequeue for caching remote message that need to be sent to other worker later. To the aim that improving the efficiency of fetching local message , localmessagequeue storing message as object instead of byte array

which allow fast access message object. However , remotemessagequeue storing message as byte array which can make maximum compression of message result in speeding up message sending . When the messages on a worker accumulate to a pre-specified threshold size (e.g. 512 k), a thread would be initiated to send the messages to their destination workers through *natty*. If a worker has processed all its received messages, however, it has to periodically check the remote message queue provided that the program does *not* terminate. In graphU , if user specify the priority , localmessagequeue is used *TreeMapObjMessagesStore* which consists of treemap data structure and storing message as object .

4.3 CONVERGENCE CONTROLLER

Convergent condition is used to decide whether algorithm is terminated .In principle, a DFA-G program can be deemed to terminate if there does not exist any running message. Due to asynchronism, the fact that all the workers have voted to halt does not necessarily mean that the program has terminated: there may still exist the messages in transit. In GraphU, the function of converge controller is implemented on the zookeeper. A zookeeper records the status of each worker, including whether it has voted to halt and its respective numbers of sent and received messages. Every time , master accumulate all handled message that other worker report to zookeeper and detect whether total number of sending messages are equal to the total number of receiving messages . Once a worker currently no local messages and remote message to compute , it will report to zookeeper and detect if master has written finished label to zookeeper . This reporting event can trigger master to calculate the newest statistics and redetermine if the running program converge . A DFA-G program can be determined to terminate if the two following conditions are satisfied: 1) all the workers have simultaneously voted to halt; 2) the total number of sent messages are equal to the total number of received messages.

Sometimes there is an algorithm that does not converge , for example A_b^1. Although each time master need to re-update statistics , messages in every worker are endless . Thereby worker always report to master , master always redetermine if it has converged . Asynchronous DFA can converge as long as that each vertex has performed update operations in an finite number of times and converged to a fixed value v_j^*. Then , DFA-G programming model can enable to guarantee that asynchronous programs will converge to the same fixed point as synchronous programs.

5 PRIORITY SCHEDULING STRATEGY

Scheduling Engine is critical part of GraphU . A optimal processing sequence of vertices may significantly accelerate convergence of the computation . Our system provide fine-grained scheduling at the same time can guarantee the consistency of the results . We implemented three different policies : scheduling based on status (S-S), scheduling based on message size (S-M) ,scheduling based on vertex value (S-V) . User can specifies the scheduling policy by the command line -sPrio,-mPrio ,-vPrio or write into the configuration file . If user do not define any schedule strategy ,then all vertices are scheduled at a time until there are no vertex in queue .

S-S scheduling policy. S-S is an important schedule policy , through specifying some status take precedence can make more vertices turn into terminal status as soon as possible or turn into status which can absorb more messages . S-S scheduling policy selected vertices based on *getMaxStatePriorVertexs* which is used to sort vertices by state priority that user specified .

S-V scheduling policy . For S-V scheduling policy , there have two parameters (max or min , dPercent) need users to specify , one is to make sure scheduling with a maximum or minimum value , the other parameter dPercent is how many vertices that fetch out one time.S-V scheduling policy selected vertices based on *getLimitValuePriorVertexs* which first sort vertices by vertex max or min value and then fetch only dPercent of vertices that need to scheduled next time .

S-M sheduling policy . There have two parameters (max or min , dPercent) need users to specify , one is to make sure scheduling with a maximum or minimum number of messages , the other parameter dPercent is how many vertices that fetch out one time. S-M scheduling policy used *getMaxorMinMsgCntPriorVertexs* function to get scheduling vertices next time .*getMaxorMinMsgCntPriorVertexs* sort vertices by the max number of message or the min number of message , and selected only dPercent of vertices that need to scheduled next time .

6 Related Work

Much of the existing work on graph processing systems is mainly based on two kinds of iterative process . One is conventional synchronous iterative systems based on BSP(eg , pregel[1],GraphX[10],Giraph[2],Hama[11]) , and the other is asynchronous iterative systems (eg,GraphLab[12],GraphHP[4],GraphUC[3],Blogel[13]). Synchronous iteration composed of many supersteps ,in each superstep , all vertices are scheduled at most one time and each time handled messages which sending in last superstep . Although simplified programming interface and can ensure consistency of algorithm results, BSP synchronous systems also have defects on global synchronization barriers which resulting in much time waiting for the slowest worker. Therefore ,researchers proposed asynchronous iterative systems . Comparing with synchronous iteration systems , asynchronous systems to some certain extent removing the synchronization barriers but itś programming model do not support multi-platform feature.Besides , they still have limitations on user-friendliness and performance optimization.Thereby , we have proposed a unified programming model,named DFA-G(Deterministic Finite Automaton for Graph processing), which has been published . While all the current synchronous systems and asynchronous systems cannot perfectly support this unified programming model . These systems either reduced the frequency of synchronization (eg,GraphHP ,Grace[8],HAGP[14]) ,which extend vertex-centric computation model to similar block-centric computation model or reduced synchronization granularity (eg.Graphlab,PowerGraph[15])using read-write locks to implement synchronization between mirror vertex and master vertex avoiding multiple adjacency vertex updating the same vertex .

In the aspect of complexity analysis. [6]proposed two graph join operators for scalable graph processing in MapReduce, which a wide range of graph algorithms can be redesigned with smaller communication cost and memory consumption . Although it take complexity of algorithm into consideration , it is still a synchronous platform , and not suitable for more

Table 7.1: 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**

complex algorithms .

7 Evaluation and Demo Plan

We empirically evaluate the performance of different DFA-G programs for Bipartite Matching (BM) and Pattern Matching (PM) on real graphs. We have implemented the three DFA-G automatons shown in Figure. 3.1 for BM. For PM, we have implemented the two DFA-G automatons shown in Figure. 3.2. 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 [9] for the effect of priority setting on DFA-G's execution performance.

The performance comparison between different DFA-G programs are presented in Table. 7.1. 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 pre-

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

Table 7.2: 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**

Table 7.3: Performance comparison of different vertex value priority on SSP.

Priority Norm	Parameter		Performance	
	Max or Min	Ratio	runtime(s)	No.of Messages(mil)
vprio	Min	1.0	38.724	41.500
vprio	Min	0.9	29.739	12.592
vprio	Min	0.8	25.256	9.738
vprio	Min	0.7	23.595	8.134

sented in Table. 7.2. 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.

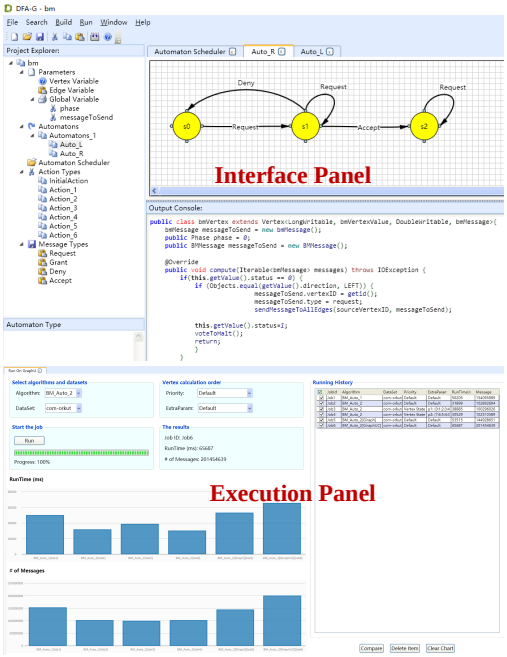

Figure 7.1: The demo system of GraphU

In this section , in order to verify the impact of priority , we take the other two classical algorithms as experimental programs, MSF[16] algorithm(Minimum Spanning Tree)and SSP[17] algorithm(Shortest Path) . Minimum Spanning Tree(MSF) is using to find conjoined trees in

Table 7.4: Efficiency evaluation on MSF algorithm.

MSF	Status Priority			
	null	0:1:2:3	3:2:1:0	2:0:1:3
No.of Messages(mil)	196.197	183.882	191.570	188.094
runtime(s)	48.025	43.267	46.207	45.904

a connected graph.The algorithm first finds 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. ShortesPath(SSP) is to find the minimum weight of all reachable path from the source vertex to the other vertex, it is widely used in many fields . The exprimental results of MSF and SSP with different priorities on GraphU are as follows: 7.4 , 7.3, , the dataset used roadNY[18] and road-NE[18] .We can see different priorities will result in different running performance , a optimal vertices computing sequence could reduce message and accelerate convergence . In the table 7.4 , 0:1:2:3 means the priority of a vertex in status0 is 0,priority of status1 is 1,priority of status2 is 2 and priority of status3 is 3 . Thus , we can see among these three kinds of priority assignments , 0:1:2:3 is the better one which produce less message. However, in SSP ,we use priority by vertex value , in table 7.3 "vprio" signify using priority by vertex value . From above tables about SSP , we can see a smaller value of vertex should take a higher priority , this will reduce unnecessary message delivery . However, to some certain extent , once each time the number of scheduled vertices that less than a certain value will conversely produce more cost in selecting the vertex that needs to be scheduled .

Demo Plan. The demo system of GraphU, whose screenshots are presented in Figure. 7.1, 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," in *http://giraph.apache.org/*, 2017.

[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.

[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 Workshops (ICDMW)*, 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] 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 *2013 IEEE International Conference on Big Data*, 2013, pp. 403–411.

[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] http://www.wowbigdata.cn/GraphU/demo.html.

[10] D. C. Reynold S. Xin, "Graphx: Unifying data-parallel and graph-parallel analytics," 2014.

[11] "Apache hama," in *http://hama.apache.org/*, 2017.

[12] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," vol. 5, no. 8. VLDB Endowment, 2012, pp. 716–727.

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

[14] 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.

[15] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs." in *OSDI*, vol. 12, no. 1, 2012, p. 2.

[16] S. Chung and A. Condon, "Parallel implementation of bouvka's minimum spanning tree algorithm," in *Parallel Processing Symposium, 1996., Proceedings of IPPS'96, The 10th International*. IEEE, 1996, pp. 302–308.

[17] C. E. L. T. H. Cormen, "Introduction to algorithms," in *3rd edition*, 2009.

[18] http://www.dis.uniroma1.it/challenge9/download.shtml.