GPSA: A Graph Processing System with Actors

Jianhua Sun, Dongwei Zhou, Hao Chen, Cheng Chang, Zhiwen Chen, Wentao Li
College of Computer Science and Electronic Engineering
Hunan University
Chang Sha, China
Email: {jhsun,dwzhou,haochen,chengchang,zwchen,wentaoli}@aimlab.org

Ligang He
Department of Computer Science
University of Warwick
Coventry, U.K.
Email: liganghe@dcs.warwick.ac.uk

***Abstract*—Due to the increasing need to process the fast growing graph-structured data (e.g. social networks and web graphs), designing high performance graph processing systems becomes one of the most urgent problems facing systems researchers. In this paper, we introduce GPSA, a single-machine graph processing system based on an actor computation model inspired by the Bulk Synchronous Parallel(BSP) computation model. GPSA takes advantage of actors to improve the concurrency on a single machine with limited resource. GPSA improves the conventional BSP computation model to fit in the actor programming paradigm by decoupling the message dispatching from the computation. Furthermore, we exploit memory mapping to avoid explicit data management to improve I/O performance. Experimental evaluation shows that our system outperforms existing systems by 2x-6x in processing large-scale graphs on a single system.**

***Keywords*-Actors; Graph Computing; Bulk Synchronous Parallel;**

I. Introduction

Graph algorithms are becoming increasingly important for solving problems in scientific computing, data mining, and other domains such as social networks and web graphs. There has been an increasing interest in distributed graph processing system, such as Pregel [11], GraphLab [10], PowerGraph [3], GPS [14], Mizan [8]. These distributed systems allow developers to write applications to process large-scale graph data with high performance.

Nowadays, although distributed computation resources are more easily accessible than ever before, however, processing large-scale graphs with distributed systems still remains challenging. In distributed systems, one of the main problems is load imbalance caused by partitioning large graphs into small partitions to fit in a single node. Since some distributed systems are based on the Bulk Synchronous Parallel (BSP) [18] Model , imbalanced workload distribution among computation nodes would affect the overall efficiency. Another issue is the communication latency. In distributed systems, different computation nodes need to exchange data or state via messages to communicate with each other, and the communication latency also should be considered. Besides, from the user's perspective, developing, debugging, and optimizing applications on distributed systems is difficult because the user needs to be skilled at managing and tuning a distributed system in a cluster, which is still a challenging job for the ordinary user. And distributed systems need many machines in a cluster, which brings both money and extra energy cost.

Recently, some graph processing engines that focus on exploiting single machine performance have been proposed to address the problems of distributed graph processing systems. Graphchi [9] is a disk-based graph processing engine running on a single machine. As graph processing often exhibits poor locality of data access, GraphChi introduces a novel mechanism called Parallel Sliding Windows (PSW) to alleviate the issue of random accesses to improve the I/O performance, and GrapChi shows comparative performance with most of the representative large-scale distributed graph processing systems. TurboGraph [4] inspired by GraphChi focuses on improving parallelism by overlapping the CPU and I/O processing with a novel concept pin-and-slide. But it is designed specifically for solid state disks (SSD) to obtain high performance. X-Stream [13] is a graph processing system that is differentiated by its edge-centric and scatter-gather model, separating the process into two phases, scatter and gather. X-Stream supports both in-memory and out-of-core graphs on a single machine.

Although existing single machine approaches have demonstrated the effectiveness with reasonable performance, they cannot fully exploit the capabilities of modern multi-core systems. Through experiments, we found that GraphChi exhibits poor utilization of the CPU on multi-core systems, while X-Stream shows poor scalability. In this paper, we present a new vertex-centric graph processing model based on the observation that, in conventional vertex-centric programming model, the computing and message dispatching procedure are executed sequentially [13]. In the BSP model, the updated value is not visible to its neighbors until the next superstep, which means the message sending procedure has no relevance to the computing procedure. Based on this observation, we present GPSA, a graph processing system based on actors, which can exploit the capabilities of multi-core systems as much as possible. We first decouple the computing procedure from the message dispatching, with which we can overlap the two processing procedures and execute them in parallel. In addition, we leverage the memory mapping mechanism provided by the operating system to handle I/Os, which not only achieves better performance but simplifies system design.

We evaluate our system by comparing it with the state-of-the-art single machine systems including GraphChi and

0190-3918/15 $31.00 © 2015 IEEE
DOI 10.1109/ICPP.2015.80

X-Stream. Our experiments show that GPSA is about 3x-4x faster than GraphChi. Compared with X-Stream, we show that GPSA is not only faster but more scalable than X-Stream.

The rest of the paper is organized as follows. Section 2 describes the background and our motivations. In section 3, we introduce the actor programming model. The new BSP model with actors is introduced in Section 4. In section 5, We describe implementation details and evaluates our work by comparing with GraphChi and X-Stream in section 6. At last, we describe related work in Section 7 and conclude in section 8.

II. Background

In this section, we first introduce the problems in parallel programming. Second, we present the prominent features of actor-based programming and brief the *Kilim* actor framework. Finally, we review the BSP computation model.

A. Parallel Programming

Since the multi-core has become the main architecture of the modern computer systems, the way of exploiting its computation capability is to develop applications using multi-threaded parallel programming model. This parallel computing paradigm is based on how to manage and control concurrent accesses to the shared mutable states, which requires explicit synchronization to avoid potential concurrency bugs such as deadlocks. Manipulating shared, mutable states via threads makes it hard to write correct and scalable applications and difficult to predict the behavior of threads at runtime. Java provides shared memory threads with locks as the primary form of concurrency abstractions. However, shared memory threads are heavyweight and suffer from performance penalties incurred by context-switches.

B. Actor

The actor programming model takes a different approach to solving the problem of concurrency, by avoiding the issues caused by shared memory, threads, and locks. Actors encapsulate data and code, and communicate with explicit messages. In this model, all objects are modeled as independent computational entities that only respond to the messages received, and there is no shared state between actors. Actors do not change their state until they receive an explicit message. Actors typically run in parallel. There are some principles for the actor model: (a) Immutable messages are used to communicate between actors. (b) Each actor has a mailbox for the incoming messages. (c) Messages are passed asynchronously. It means that the sender does not wait for the message to be received and can go back to its execution immediately. (d) Communication between the receiver and sender is asynchronous, which means that they can execute in different threads.

The standard actor semantics provides encapsulation, location transparency, fair scheduling, locality of reference, and transparent migration. These properties enable simplified design and performance improvement, and make applications scalable. Encapsulation means that an actor cannot directly access the internal state of another actor, and the messages transfered between actors should have call-by-value semantics. Location transparency indicates that the actual location of an actor has no influence on its name and one actor does not know the address of another actor. Fair scheduling means that no actor can be permanently starved. Transparent migration is defined as the ability of a computation to move across different nodes including both code and execution state. With the actor model, it is more flexible to build highly concurrent and scalable applications.

C. Actors in Kilim

In this paper, we use Kilim as our actor programming framework. Kilim [16], [17] is an actor-based library written in Java. In Kilim, *Actors* are represented by the Kilim type of *Task*. *Task*s are lightweight threads and communicate with other*Tasks* via *Mailbox* that can accept "messages" of any type. *Task* can send messages and even customize message types. All the operations performed on actors are bound via method signatures. For example, The *Pausable* signature implies that the bound function can be run concurrently. Kilim relies on a static code modifier called *Weaver* to realize the actor interface by instrument the Java byte-code. Pausable methods throwing clauses are processed at runtime by a scheduler, which is part of the Kilim library and manipulates a certain number of kernel threads. It is able to leverage this pool for a higher number of lightweight threads, which can switch context and start up quite fast. Each thread's stack is automatically managed. The actor model makes it easier and safer to write asynchronous-acting objects that depend on similar objects.

D. Vertex-centric Graph Processing

Pregel is the first distributed vertex-centric programming model introduced by Google. As shown in Figure 1, the computation in Pregel consists of a sequence of iterations (*supersteps*). And during a superstep, Pregel invokes the *compute* function defined by the programmer for each vertex, conceptually in parallel. This *compute* function specifies the behavior at a single vertex V and a single superstep S. Furthermore, it can read messages sent to V in superstep $S-1$, send messages to its neighbor that will be received at superstep $S+1$, and modify the value of V. In addition, between two adjacent supersteps, a barrier is imposed to synchronize that all vertices finish processing messages.

III. Motivation

Actor model is conceptually similar to the vertex-centric programming model. Actors communicate with each other via messages. However, existing vertex-centric systems use thread as the main execution unit,like GraphChi , which limits the concurrency level. In addition, the vertex-centric model needs to maintain a large number of messages in persistent storage for the next superstep, which makes the single machine approaches hard to scale. Thus, single

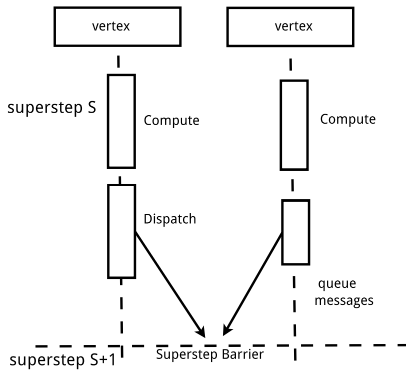


Figure 1: Vertex-centric model

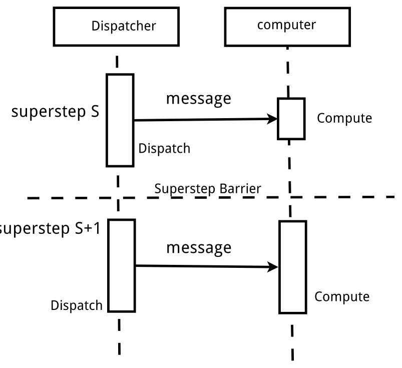


Figure 2: New BSP model with actors

machine approaches put a lot efforts on optimizing I/O performance. In this section, we first discuss the inefficiencies of the vertex-centric model. Then, we motivate the work of this paper.

A. *Inefficiency of Vertex-centric Model*

As shown in Figure 1, in superstep *S*, for a single vertex *V*, the vertex-centric model invokes the user-defined method *compute* to execute the computing procedure first and then executes the dispatching procedure to send update to its neighbors, and finally reaches the barrier. The whole process is executed sequentially.

First, We observed that messages for the next superstep *S+1* will not be processed in the superstep *S* until the computing procedure of the superstep *S+1*. Those messages intended for the next superstep have to be stored somewhere, indicating extra memory consumption or extra I/O operation.

Second, the computing procedure has a strong dependency on the messages, however, the relationship between two procedures acrosses two supersteps. Considering the synchronization at the end of each superstep, the processing seems to be discrete.

B. *Motivations*

Based on the above observations and analysis, we consider the following principals when designing an actor-based graph processing system.

- Given the flexibility of the actor programming model and its similarity with the vertex-centric graph processing, the actor model is a natural fit for our purpose as compared to the conventional thread-based solutions.
- By exploiting the potential concurrency among message dispatching and computing using the actor model, and the optimization on disk I/Os, we expect performance improvement in single machine based graph processing.
- Actor-based graph processing can not only benefit multi-core systems but also be directly applicable to distributed systems, although our focus in this paper is leveraging actors to accelerate graph processing in a single multi-core system.

IV. System Design

In this section, we first introduce our new BSP model and the work flow of our system. Then, we analyze the data access behavior of the new model. Next, we describe the data organization in GPSA. Finally, we detail how the operations of message dispatching and value updating are performed.

A. *New BSP Model with Actors*

The new model is an asynchronous computation model by overlapping the dispatching procedure and computing procedure. Figure 2 depicts the new actor-based BSP model, in which the vertex passively processed by threads are replaced by active light-weight actors. And the relationship between two procedures is placed in one superstep to execute in parallel. We abstract two kinds of actors according to the two procedures, which are dispatching actors and computing actors. From Figure 2, the dispatching actors send messages to the computing actors and the computing actors are responsible for processing these messages.

In our actor-based BSP model, the reason why we replace threads with actors is that actor can not only achieve higher concurrency but also have more functional semantics than vertex. First, in traditional vertex-centric model, the basic execution unit is thread, and it has to keep the integrity of the execution on a vertex, causing inefficiency as mentioned above. Second, vertex-centric means that vertex is the main functional unit and responsible for all the message processing. The real processing of a vertex is scheduled one by one by threads. However, the computing actor is responsible for all the message processing and the processing is message-driven, which provides more flexibility.

As a result, there are two different roles in our computation model, including dispatching actors and computing actors as illustrated in Figure 3. Dispatching actors send messages to compute actors that conduct the real computation based on the input from the received messages. In our model, the dispatching and computing actors both are basic execution unit that is schedulable by the runtime system, and all the messages communicated among vertices in the vertex-centric model are now passed via actors.

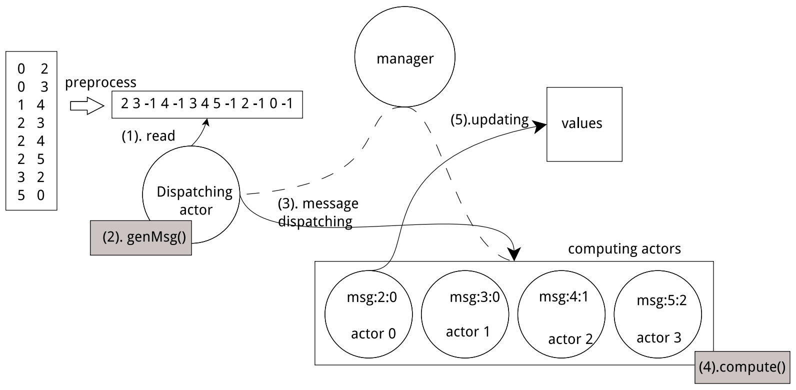


Figure 3: **Work flow overview**

In the traditional BSP model, there always is a superstep 0 that only sends messages but does not process messages at all. However, in the new model, we can start the processing of messages at once upon the arrival. And at the beginning of a new superstep, when an actor needs to send a message, it does not have to wait for the computation of the current superstep to finish, because the computation for the current iteration has already been finished in the last iteration.

B. Data Access Behavior

In our model, we replace vertices with actors, and the message passing between vertices is transformed into the communication between actors, which results in different data access behaviors as compared to the traditional BSP model. First, the new model does not need to queue a large number of messages into persistent storage, because the dispatching and computing actors together form a classical producer-consumer pattern. The computing actors listen to the event of message arrival. Upon such an event, the computing actors will be scheduled to process the incoming messages immediately.

Second, the vertex values should be accessed both randomly and efficiently. Since for a computing actor it does not care about how message comes and only concerns values conveyed by messages and what to do with values. To process a message, it first needs to obtain the value of the destination vertex *V*, and then invoke the user-defined method *compute* and update the vertex value. A message usually contains the destination and value. Actors respond to messages. However, as messages are delivered randomly, random access would occur, which should be avoided to guarantee higher performance. Once actors receive a message, the value of the specified vertex should be available not matter where it is stored (in memory or external storage).

C. Disk I/O

Both GraphChi and X-Stream need to write a large amount of data to the disk, which necessitate specific optimizations to improve I/O performance. GraphChi introduces an asynchronous model called Parallel Slide Window by partitioning the graph into intervals. X-Stream also presents an asynchronous I/O optimization to achieve better performance.

The new BSP model takes advantage of actors, and it does not need to buffer messages to the disk for the next superstep. The only challenge is that both the dispatching and computing actors need to access the vertex values efficiently given the randomness of this operation which is keen to avoid in other system. We leverage the memory mapping offered by modern operating systems to manage I/O requests for vertex values. Furthermore, the edges in fact are processed by dispatching actors sequentially from disk.

D. Data Organization

In the new BSP model, it needs to store two copies of values, because the dispatching and computing actors handle different values. As mentioned above, we take advantage of memory mapping to access the vertex values that are stored in a single file according to numeric *id*s, and the two copies of the value are next to each other. The offset of the value for vertex V can be calculated with $|V| * sizeof(Val)$.

To save memory consumption and improve I/O performance, we store graphs in Compressed Sparse Row (CSR) format. The CSR format arranges the edges in a big array sorted according to its source vertex *id* and separated by a specified symbol. For example, in Figure 4, vertex 0 has two out-edges pointing to vertex 2 and vertex 3, and the symbol -1 indicates the end of the edge list of the current vertex.

In fact, in our model, actors only handle values and messages. So it is possible to process the original edge list without a preprocessing procedure. However, it is more flexible to structure the graph using the CSR format. For example, in PageRank algorithms, the out-degree of a vertex matters when generating a message. To improve I/O performance without an extra lookup, we can store this information in the CSR format , as shown in Figure 4.

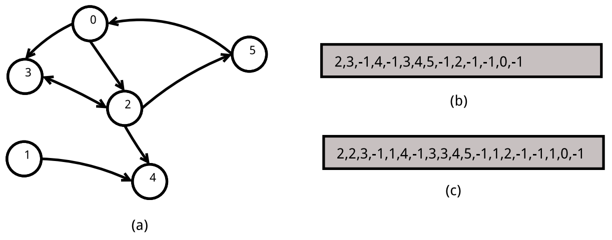
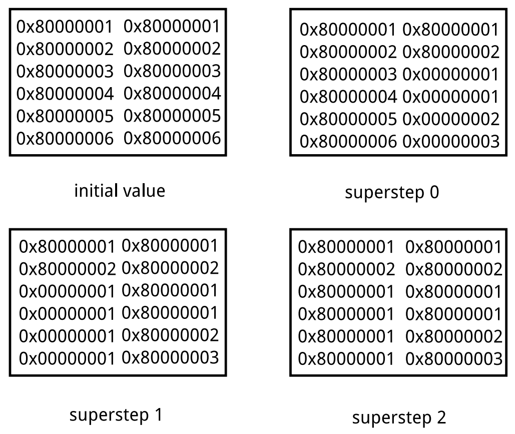


Figure 4: **CSR example**

(a) An example graph (b) A CSR file without vertex degrees (c) A CSR file with vertex degrees



```
0x80000001  0x80000001
0x80000002  0x80000002
0x80000003  0x80000003
0x80000004  0x80000004
0x80000005  0x80000005
0x80000006  0x80000006
```

initial value

```
0x80000001 0x80000001
0x80000002 0x80000002
0x80000003 0x00000001
0x80000004 0x00000001
0x80000005 0x00000002
0x80000006 0x00000003
```

superstep 0

```
0x80000001 0x80000001
0x80000002 0x80000002
0x00000001 0x80000001
0x00000001 0x80000001
0x00000001 0x80000002
0x00000001 0x80000003
```

superstep 1

```
0x80000001  0x80000001
0x80000002  0x80000002
0x80000001  0x80000001
0x80000001  0x80000001
0x80000001  0x80000002
0x80000001  0x80000003
```

superstep 2

Figure 5: **Value updating**

E. Message Dispatching

A message consists of the value and the identifier of the destination vertex. How to generate message values varies with different applications. For example, in the PageRank algorithm, the value of a message is related to both the out-degree and the vertex value. While it only depends on the vertex value in the BFS algorithm. Therefore, it is delegated to the user to implement the logic of how to generate a message according to the vertex value, out-degree, and even the edge weight, as shown by the function *genMsg()* in the Figure 3. First, if a vertex value has been updated in the last iteration, it will invoke the user-supplied function to generate the message value. Next, the dispatcher packs the value together with an integer identifier (*id*) and locates the computing actor that will process this message according to the destination *id*. At last, the message is sent to the desired computing actor.

F. Updating

As shown in Figure 3, after invoking the user-defined *compute()* method, the vertex value should be updated. We store two copies of different values for the same vertex. One copy is generated during the last superstep, while the other is the result of the superstep before the last one. Values from the last superstep will be used in dispatching, and others will be overwritten by the update operation. In fact, all the values are stored in a file sequentially; from the perspective of vertex *id*, it looks like two columns as shown in Figure 5.

For a computing actor, two main problems may occur. First, the values in the memory-mapped file are organized in two columns for each vertex. One column is used for sending messages, and the other is used for value updating. After a superstep, the values of the two columns will become different. So if the computing actor fetches the value randomly, it will get the wrong data because the validate value is stored in the message-sending column actually. So there is a need for the computing actor to figure out the first message of a vertex and fetch value from the message sending column then write it into the updating column.

Second, as mentioned in the last section, if a vertex value is not updated, the dispatcher will skip it. Messages come randomly, so it is difficult to identify whether a value is updated or not. Although the computing actor knows the presence of updating, it has no idea of what to do with it. In order to convey the updated information to the dispatcher, we set the highest bit of the vertex value to 1. At first, all the values will be set. During the computation, if a value has been updated, the highest bit will be reset to 0. Otherwise, if a message is the first message of a vertex, a negative value will be written. After a dispatcher finishes processing, it will invalidate the value of the current vertex by setting its highest bit to 1.

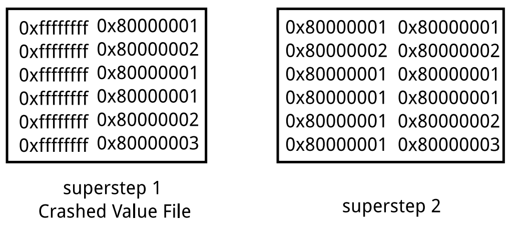
For example, in Figure 5, values on the two columns are initialized to the same using the user-supplied function with the highest bit set to 1. In superstep 0, dispatchers read values from the left column, and compute actors update values on the right column. If a value is updated, its highest bit is set to 0, like the 0x00000001 and 0x00000002. In superstep 1, the dispatchers read values from the right column and computing actors update the left column. In each superstep, the dispatchers can disable the value by setting the highest bit to 1.

G. Lightweight Fault Tolerance

Fault tolerance is important to guarantee the reliability of the long-running graph computation. Although distributed graph processing systems have built-in fault tolerance support, in single-machine systems such as GraphChi and X-Stream, it is complicated (if not possible) to implement fault tolerance, because of the requirement of recording huge volume of program states to the disk, which may incur unacceptable performance degradation.

In GPSA, due to our asynchronous design, in a superstep, there is always an immutable column, which ensures that there is always a valid copy of result of the last iteration stored. If the system crashes within the current iteration, we can recover the system state from the latest successful iteration.

For example, as shown in Figure 5, if the processing fails in superstep 1, and the states of the vertex value file is shown in Figure 6. In superstep 1, the right and left columns are used for dispatching and updating respectively. Therefore, the right column contains the valid data since the last successful superstep, and we can recover the system state from the crashed data file.



0xffffffff	0x80000001
0xffffffff	0x80000002
0xffffffff	0x80000001
0xffffffff	0x80000001
0xffffffff	0x80000002
0xffffffff	0x80000003

0x80000001	0x80000001
0x80000002	0x80000002
0x80000001	0x80000001
0x80000001	0x80000001
0x80000001	0x80000002
0x80000001	0x80000003

superstep 1
Crashed Value File
superstep 2

Figure 6: **Crashed value file.**

V. Implementation

We have implemented GPSA in Java based on the actor framework Kilim.

A. Overview

We assume that the vertices are labeled from 0 to $|V|$. GPSA has two types of actors, the manager and the worker, and consists of four main modules including preprocessing, manager, dispatcher, and computing worker. The format of input data is text-based edge list or adjacency graph. As shown in Figure 3, the graph is first transformed into the binary form CSR representation. After preprocessing, the manager starts to initialize the data file that stores the vertex values by invoking an user-supplied *initialize* function, and then the manager assigns a range of vertex *id*s to the dispatcher worker and edges to the computing worker. Third, the manager creates the computing worker and the dispatchers in turn. At last, all workers enter into their execution loops.

There are different ways to read the memory-mapped files to balance load among workers. For example, for convenience, the vertices can be read by the dispatching worker with a simple mod algorithm. For efficient,we can assign vertices to the dispatcher worker by the average edges to ensure that every dispatcher worker sends exactly the same number of messages. In addition, there are also different strategies to deliver a message to a specific computing worker. The easiest way is an average assignment by mod according the vertex id. In this way, every single computing worker will have no conflict with others when updating. However, it may cause the unbalanced workload, because the new message-driven model is still a BSP model, the unbalance may be an overhead. To be more flexible, we provide interfaces for developer to substitute the default implementation.

B. Preprocessing

Although our system can process the original binary edge-list input with our new model. We still recommend a preprocessing phase, with which we can arrange the edges of a vertex close to each other to reduce the time of reading data from the memory-mapped file. If the input graph is in adjacency format, we can just write the destination vertex *id* into the memory-mapped file. But with the edge-list format, an extra sorting operation is needed to transform it into the adjacency format.

C. The Manager

The responsibilities of the manager actor include initializing values, coordinating the computation, handling exceptions, and monitoring workers. Each manager keeps track of the states of its own workers. As shown in Algorithm 1, when the computation begins, the manager will send the workers an ITERATION_START command to signal them to start execution immediately. When a dispatcher worker finishes all its work in the current iteration, it will inform the manager with a DISPATCH_OVER command and wait for another ITERATION_START command for the next iteration. The manager is informed to enter into the next iteration when it receives the DISPATCH_OVER from all the workers, and a COMPUTE_OVER command is then sent to its computing workers, which will reply a COMPUTE_OVER back to the manager. At last, if the computation is about to end, a SYSTEM_OVER command is issued to kill all the worker actors.

Algorithm 1 Manager Execution Loop

```
procedure EXECUTE($endIte$)
    $dispatchers, computers, mailbox, signal$
    $counter \leftarrow 0$
    $currIte \leftarrow 0$
    while $currIte < endIte$ do
        for $dispatcher \in dispatchers$ do
            $dispatcher \leftarrow ITERATION\_START$
        end for
        while $(signal \leftarrow mailbox.get()) == DISPATCH\_OVER$ do
            $counter \leftarrow counter + 1$
            if $counter == dispatchers.length$ then
                $counter \leftarrow 0$
                $break$
            end if
        end while
        for $worker \in computers$ do
            $worker \leftarrow COMPUTE\_OVER$
        end for
        while $(signal \leftarrow mailbox.get()) == COMPUTE\_OVER$ do
            $counter \leftarrow counter + 1$
            if $counter == computers.length$ then
                $counter \leftarrow 0$
                $break$
            end if
        end while
        $currIte \leftarrow currIte + 1$
    end while
end procedure
```

D. *The Workers*

The worker actors including the dispatching and computing worker are the basic execution unit, and play the most important role in the system. Dispatchers are responsible for reading edge values and sending vertex update messages to computing workers. The computing workers listen to the message event. If there are messages arrived, the computing worker will be notified to get the messages from its own mailbox and conduct computation on the messages with the user-supplied *compute()* method.

Algorithm 2 Dispatcher Execution Loop

```
1: procedure EXECUTE
2:     $signal \leftarrow mailbox.get()$
3:     while $signal! = SYSTEM\_OVER$ do
4:         if $interval! = null$ then
5:             $reset()$
6:             while $curoff < endoff$ do
7:                 $val \leftarrow getValue(sequence)$
8:                 if $isHighestBit(val) == 1$ then
9:                     $skip(sequence)$
10:                end if
11:                if $isHighestBit(val) == 0$ then
12:                    $vid \leftarrow readEdge(curoff)$
13:                    while $curoff < enfoff$ do
14:                        if $vid == -1$ then
15:                            $break$
16:                        end if
17:                        $msg \leftarrow genMsg()$
18:                        $dispatch(vid, msg)$
19:                    end while
20:                    $setHighesetBitTo1()$
21:                end if
22:            end while
23:        end if
24:        $notifyManager(DISPATCH\_OVER)$
25:        $signal \leftarrow mailbox.get()$
26:    end while
27: end procedure
```

Inside the dispatcher, the data structure called *interval* maintains the *id*s between the first to the last vertex, and the addresses of the starting and ending offset in the memory-mapped file. According to the *id* sequence, the dispatcher worker can identify which vertex it is processing. The offset indicates the position of the next edge. As shown in Algorithm 2, the dispatcher loops until the signal SYSTEM_OVER is received. After getting the vertex value, the dispatching work invokes the *genMsg* method to generate a new message to be sent to the computing worker, if the value of the current vertex is updated during the last super (the highest bit is 0).

The computing worker is responsible for processing messages and updating vertex values. As shown in Algorithm 3, upon a message arrival, the computing worker checks if the message is a signal or a normal message. For a normal message, the destination *id* and message value are extracted. According to the *id*, it fetches the vertex value from the memory-mapped file and invokes the *compute* method. At last, new value (if any) is written to the memory-mapped file.

Algorithm 3 Compute Execution Loop

```
1: procedure EXECUTE
2:     $msg \leftarrow mailbox.get()$
3:     while $signal! = SYSTEM\_OVER$ do
4:         if $msg == COMPUTE\_OVER$ then
5:             $notifyManager(COMPUTE\_OVER)$
6:         else
7:             $to \leftarrow msg.dest()$
8:             $msgVal \leftarrow msg.val()$
9:             $val \leftarrow getVal(to)$
10:            $newVal \leftarrow compute(val, msgVal)$
11:            if $newVal! = val$ then
12:                $update()$
13:            end if
14:        end if
15:    end while
16: end procedure
```

VI. EXPERIMENT

We first introduce the testing environment and datasets used in the experiments. Then, we compare our work with GraphChi and X-Stream, two state-of-the-art systems, by measuring the execution time of three representative graph algorithms and the CPU utilization.

A. *Setup*

All the experiments are performed on the same machine equipped with 32 cores (8GHZ Intel i7), 16GB memory, and 1TB disk (7200RPM). The operating system was Ubuntu 12.04. We selected four graphs with different size: the Google network with 5 million edges, the soc-pokec with 30 million edges, the LiveJournal with 69 million edges, and the twitter-2010 with 1.4 billion edges, as summarized in the Table I.

Name	Nodes	Edges
google	875,713	5,105,039
soc-pokec	1,632,803	30,622,564
soc-liveJournal	4,847,571	68,993,773
twitter-2010	41,652,230	1,468,365,182

Table I: Graphs used in experiment

B. *Performance*

We compare with GraphChi (0.2.6 C++ version) and X-Stream using the default configurations. Each test is run under the same configuration for 3 times and calculate the average. Because of the different implementation of three approaches, we choose to compare the average an average elapsed time of five supersteps. From Figure 7 to Figure 10,

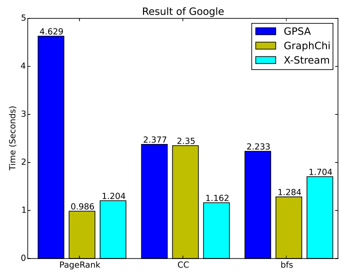


Figure 7: Google Graph

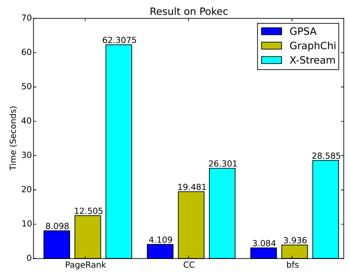


Figure 8: Pokec Graph

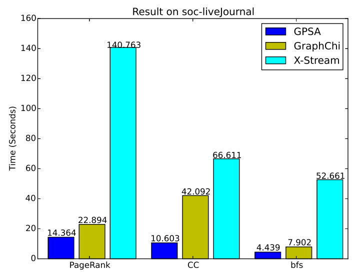


Figure 9: journal Graph

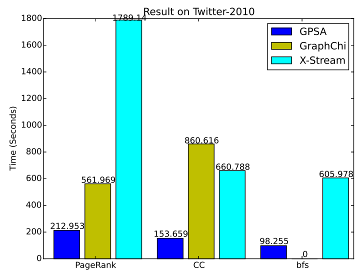


Figure 10: Twitter-2010 Graph

the performance of the three algorithms on different graph datasets are depicted.

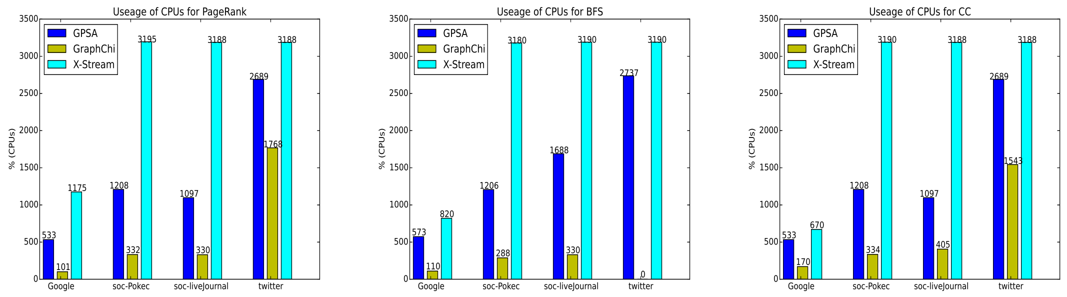
Result on Google Graph: Figure 7 shows the running times about the Google graph, and we can see that X-Stream performs best in finding connected components. While in running PageRank and BFS, GraphChi outperforms others. Of all the three algorithms, GPSA is slower than GraphChi and X-Stream. For PageRank, GPSA is about 4x slower than GraphChi and X-Stream. For CC, GPSA is nearly the same with GraphChi, but still slower than X-Stream. For BFS, GPSA is about 0.2x slower than GraphChi and X-Stream. This mainly because that the size of Google graph is very small. Both GraphChi and X-Stream could loaded the data into memory fully and all the updating happened in memory without much I/O read or write. Although, for GPSA, the Google graph could also be loaded into memory fully, however, GPSA is implemented with JAVA, which is may slower than GraphChi and X-Stream without program optimization.

Result on Pokec and liveJournal: Figure 8 and Figure 9 show the results on the Soc-Pokec and soc-liveJournal. In two figure, we can see that GPSA gains impressive performance. In PageRank, it is about 0.3x faster than GraphChi, 8x faster than X-Stream on soc-Pokec and 10x faster on soc-liveJournal. In connected component, GPSA is 4x faster than GraphChi and 6x faster than X-Stream on both soc-Pokec and soc-liveJournal. In bfs, GPSA is as faster as GraphChi while X-Stream performance worst. And This mainly because that both GPSA and GraphChi are vertex-centric programming model, while X-Stream is edge-centric. In a vertex centric model, it will skip the inactive vertex, which did not update, including their edges. While X-Stream iteration over each edges every superstep.

Result on Twitter-2010: Figure 10 shows the results on the Twitter-2010 graph. In bfs, we are unable to run bfs implementation provided by Aapo Kyrola on Google Code, because after preprocessing and reshard GraphChi blocked there and doing nothing at all. From Figure 10, we can see that GPSA is 2x faster than GraphChi and 8x faster than X-Stream in PageRank, 5x faster than GraphChi and 4x faster than X-Stream in connected component, and 6x faster then X-Stream in bfs. We believe our significant speedup is due to the overlapping of two procedure and the saving from memory mapping. With overlapping, the processing is performed in parallel, which reduces the average time consumption of each superstep. Besides, with CSR format data, we compress twitter graph from 26GB to 6.5 GB, which could be fully mapped into our memory, while the GraphChi will have to do reshard and X-Stream will have to do frequent I/O operation.

C. Usage of CPUs

We investigate the CPU utilization of the three approaches. The tests were conducted with all 32 cores enabled. we compare usage of the CPUs in different application on the datasets. From Figure 11, we can observe that X-Stream exhibits excellent CPU utilization with all cores fully exploited (almost 100%). However, X-Stream fails to show flexible scalability. Even with the Soc-Pokec that is a relative small dataset, X-Stream still occupies most of the CPU time. GraphChi shows the worst CPU utilization mainly because of its specific focus on I/O optimization instead of optimizing CPU parallelism. In GPSA, the CPU utilization varies according to the complexity of workloads



(a) (b) (c)

Figure 11: Usage Of CPUs

instead of compulsorily using the CPU even if the workload is low like X-Stream. We attribute this to our actor based model, which can delegate the compute and dispatch task to different actors, leading to better CPU usage as compared to thread based approaches.

VII. Related Work

Many scalable graph processing systems have recently been proposed. We survey some of the most relevant works, which can be broadly classified into single-machine and distributed approaches.

A. Distributed Approaches

MapReduce [2] is a programming model for many distributed applications. It takes a set of *Key/Value* pairs as input and produces a set of *Key/Value* pairs as output with *map* and *reduce* functions. MapReduce can also be used to perform computation on large graphs, but it could lead to suboptimal performance due to its iterative nature. GraphLab is a graph processing system focusing on machine learning algorithms. GraphLab [10] improves upon abstractions like MapReduce by expressing asynchronous iterative algorithm more compact to increase the degree of parallelism. Pregel [11], a distributed approach proposed by Google, introduces a vertex-centric computation model for large-scale graphs. However, as a distributed system, some issues still remain such as load balancing among clusters. GPS [14], [5], [19] is similar to Google's proprietary Pregle, but it provides new features including extended API with DSL support, dynamic repartitioning scheme and optimizations of distributing high degree vertices across compute node. Mizan [8] is also a Pregel-like system that aims at optimizing load imbalance. PowerGraph introduces a new abstraction that is leveraged for distributed graph placement and representation. PowerGraph exposes greater parallelism, and reduces network communication and storage costs. Kineograph [1] is also a distributed system providing extract timely insights of the graph. As Kineograph takes a stream of incoming data to construct a continuously changing graph, it can capture the relationships that exist in the data feed. In addition, the incremental graph-computation engine in Kineograph can keep up with continuous updates on the graph. Other systems such as Pegasus [6] and Gbase [7] are based on MapReduce and support matrix-vector multiplication using compressed matrices.

B. Single-machine Approaches

X-Stream [13] is a system for processing both in-memory and out-of-core graphs on a single shared-memory machine. X-Stream takes advantage of using an edge-centric scatter-gather model and streaming completely unordered edge lists rather than performing random access. Grace [12] is a in-memory, graph-aware, transactional graph management system with features such as query, search, and iterative computations. Ligra [15] is a lightweight graph processing framework that is specifically designed for shared-memory multi-core systems. GraphChi [9] is a disk-based single-machine system following the asynchronous vertex-centric programming model. GraphChi proposes the parallel sliding windows (PSW) to handle disk-based large-scale graphs and avoid random access issues. TurboGraph [4] inspired by GraphChi focuses on improving parallelism by overlapping the CPU and I/O processing with a novel concept pin-and-slide.

VIII. Conclusions

In this paper, we present a novel computation model based on actors to process large scale graphs, by decoupling the message dispatching from computation and leveraging memory mapping to mitigate the frequent random I/O accesses. With the new model, we achieve higher performance with scalable parallelism with thousands of actors, and better I/O throughout with limited physical memory. Our work shows the promise of exploiting actor programming paradigm in large scale graph processing.

Acknowledgment

This research was supported in part by the National Science Foundation of China under grants 61272190 and 61173166, the Program for New Century Excellent Talents in University, and the Fundamental Research Funds for the Central Universities of China.

References

[1] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu,F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: Taking the Pulse of a Fast-Changing and Connected World. In Proc. of the 7th ACM European Conference on Computer Systems (EUROSYS), Bern, Switzerland, 2012, pp. 8598.

[2] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In Proc. of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2004, pp. 137150.

[3] J. E. Gonzalez, Y. Low, H. Gu, et al. Powergraph: Distributed Graph-parallel Computation on Natural Graphs. In Proc. of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2012, pp. 17-30.

[4] W. S. Han, S. Lee, K. Park, et al. TurboGraph: A Fast Parallel Graph Engine Handling Billion-Scale Graphs in a Single PC. In Proc. of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), 2013, pp. 77-85.

[5] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In Proc. of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2012, pp. 349-362.

[6] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A Peta-scale Graph Mining System - Implementation and Observations. In Proc. of the 9th IEEE International Conference on Data Mining (ICDM), 2009, pp. 229238.

[7] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. Gbase: a Scalable and General Graph Management System. In Proc. of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), 2011, pp. 10911099.

[8] Z. Khayyat, K. Awara, A. Alonazi, et al. Mizan: A System for Dynamic Load Balancing in Large-Scale Graph Processing. In Proc. of the 8th ACM European Conference on Computer Systems (EUROSYS), 2013, pp. 169-182.

[9] A. Kyrola, G. Blelloch, C. Guestrin. GraphChi: Large-scale Graph Computation on Just a PC. In Proc. of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2012, pp. 31-46.

[10] Y. Low, J. Gonzalez, A. Kyrola, et al. Graphlab: A New Framework for Parallel Machine Learning. arXiv:1006.4990, 2010.

[11] 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 Proc. of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD), New York, NY, USA, 2010, pp. 135146.

[12] V. Prabhakaran, M. Wu, X. Weng, et al. Managing Large Graphs on Multi-cores with Graph Awareness. In Proc. of the 2012 USENIX conference on Annual Technical Conference (USENIX ATC), 2012.

[13] A. Roy, I. Mihailovic, W. Zwaenepoel. X-Stream: Edge-centric Graph Processing Using Streaming Partitions. In Proc. of the 24th ACM Symposium on Operating Systems Principles (SOSP) 2013, pp. 472-488.

[14] S. Salihoglu and J. Widom. GPS: A Graph Processing System. In. Proc. of the 25th International Conference on Scientific and Statistical Database Management (SSDBM), 2013.

[15] J. Shun and G. E. Blelloch. Ligra: a Lightweight Graph Processing Framework for Shared Memory. In Proc. of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2013, pp. 135-146.

[16] S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java. In Proc. of the 22nd European Conference on Object-Oriented Programming (ECOOP), Springer, Berlin Heidelberg, 2008, pp. 104-128.

[17] S. Srinivasan. Kilim: A Server Framework with Lightweight Actors, Isolation Types and Zero-copy Messaging. University of Cambridge, Computer Laboratory, Technical Report, 2010 (UCAM-CL-TR-769).

[18] L. G. Valiant. A Bridging Model for Parallel Computation. Communications of the ACM, 33(8), 1990, pp. 103111.

[19] S. Hong, S. Salihoglu, J. Widom, et al. Compiling Green-Marl into GPS. 2012.