Towards Balance-Affinity Tradeoff in Concurrent Subgraph Traversals

Yinglong Xia[1], Lifeng Nai[2], Jui-Hsin Lai[1]
[1]*IBM Research, Yorktown Heights, NY 10598, USA*
[2]*Georgia Institute of Technology, Atlanta, GA 30332, USA*
{*yxia, larrylai*}*@us.ibm.com, lnai3@gatech.edu*

***Abstract*—Graph technologies have been widely utilized for building big data analytics systems. Since those systems are typically wrapped as service providers in industry, it is critical to handle concurrent queries at runtime by incorporating a set of parallel processing units. In many cases, such queries result in local subgraph traversals, which essentially require an efficient scheduling scheme to explore the tradeoff between the workload balance and the task affinity. In this paper, we present an auction based approach for allocating concurrent subgraph traversals onto the processors. A dynamic weighted bipartite graph is built to model the affinity between subgraph traversals and processors, and the workload of processors. In particular, an edge between a task and a processor in the bipartite graph represents that the data needed by this task is likely cached by this processor. The task vertices and edges are dynamically added or removed, and the heavier edge weight represents stronger belief of the affinity. Besides, the edge weight is also governed by the current workload of the corresponding processor. We perform a parallel auction algorithm to figure out a near-optimal assignment of the subgraph traversal tasks onto the processors, which therefore addresses both the workload balance and the task affinity. The auction algorithm is performed incrementally, so as to capture the changes of the bipartite graph structure. Our experiments show the superior performance of the proposed method for various real-world use cases based on concurrent subgraph traversals.**

***Keywords*-graph; scheduling; data locality; optimization;**

I. Introduction

Graph plays increasingly important roles in big data analytics. First, a number of applications are explicitly based on graphs, such as Twitter network, Facebook graph, and co-authorship network [1][2]. Second, many analytical methods model big data as a large scale graph with properties. For example, in recommendation systems, we link customers to the products that they purchased to analyze the similarity of interest spaces of users [3]. The tasks defined on such big graphs typically are variances of local subgraph traversals. Therefore, graph technologies draw extremely high attentions in several communities, such as data mining, machine learning, social monitoring, and NoSQL databases [4].

It is fundamentally challenging to develop efficient graph technologies for big graph problems, since it requires insights of the underlying infrastructure. Nowadays, many industrial platform vendors offer concurrency capability for enterprise level big data solutions. For example, shared-nothing/disk architecture is utilized for enabling multiple database engines running in parallel on partitioned memory spaces [5]. Such infrastructure motivates us to develop efficient analytic solutions by fulfilling the following characteristics: 1) `Concurrency`: In real big data systems, there are typically a large set of queries arriving concurrently. They should be distributed to a set of parallel processing units to improve high throughput. 2) `Affinity`: Computations on a big graph can be dependent. For example, if two subgraphs are largely overlapped, traversing them on the same processor can result in improved data locality [1]. 3) `Balance`: The overall execution time of concurrent tasks is determined by the slowest process. Thus, we must ensure the workload is approximatelybalanced across parallel processing units.

This paper explores the trade-off between task affinity and workload balance in the scenario of concurrent local subgraph traversals in a big graph. By big graph, I mean either the number of vertices/edges is massive, or the amount of property/attribute associated with each vertex/edge is high. Given a big graph and a set of concurrent subgraph traversal tasks, our goal is to assign these tasks to a set of processing units, so that a task is assigned to a processing unit where the corresponding subgraph data is fully or partially cached by the processing unit; while the workload is approximately balanced across all the processing units. Note that new tasks keep on streaming in, thus the task allocation must be dynamic and adaptive.

The contributions in this paper include:

- We propose a novel scheduling method for allocating *concurrent subgraph traversals* in a large scale graph onto a set of parallel processing units. The scheduling method models affinity and workload balance using a weighted *dynamic bipartite graph*.
- We leverage data *locality-awareness* by allocating subgraph traversal tasks to affinitive processing units, in order to reduce the overhead caused by irregular data access in many graph analytics.
- We address workload *balance-awareness* by scheduling subgraph traversal tasks according to a dynamic bipartite graph, which makes the busy processors less attractive to new tasks while preserving the task affinity.
- We present an incremental parallel implementation of *auction* algorithm for finding a match in a dynamic bipartite graph. The edges selected by the match give the task assignment.

- We implemented the proposed methods and evaluated the implementation using a series of representative real-world applications involving concurrent subgraph traversals on enterprise big data platforms.

The rest of the paper is organized as follows. We discuss relevant background in Section II and present related work in Section III. In Section IV, we model the graph data locality and workload balance in concurrent subgraph traversal using a weighted dynamic bipartite graph, followed by the discussion on a parallel incremental auction algorithm in Section V.The experiments are presented in Section VI and Section VII concludes the paper.

II. Background

A *property graph* can be denoted by $G(V, E, \Theta)$, where V is the vertex set; E is the edge set and $\Theta = \{\theta | v \to \theta_v, v \in V\} \cup \{\theta' | e \to \theta'_e, e \in E\}$ represents the parameters associated with the vertices and/or edges, a.k.a. the *property* [2][4]. θ is typically a user-defined data structure which is usually organized as a hash map $\theta_v = \{m_i \to w_i\}$ with a property name m_i to a value w_i. For different vertices/edges, the property name set $\{m_i\}_{\forall i}$ can be identical or not, depending on if the graph is schema or schemaless. For example, if a vertex represents a user, then the properties can be the the user's ID, name, gender, affiliation, etc.

Graph *traversal* is the process of visiting all the vertices in a graph from a given starting vertex in a particular manner, updating and/or checking their parameters along the way. In reality, many big graph analytics only need to traverse a local subgraph, rather than the entire graph. This is known as the local *subgraph traversal*.

Here are a few examples based on subgraph traversals, some of which will be used for evaluating the proposed method in this paper.

- *SSSP with bounded length* in undirected graph is a simplified solution for finding a single source shortest path (SSSP) between two vertices, say v and u, in a given graph G, where only the paths with length less than δ is concerned. This solution performs two breadth-first search (BFS) from v and u, respectively, each searching at most $\delta/2$ hops away from the starting vertex. Once the two BFSs meet at some vertex, a shortest path is identified.
- *Naive collaborative filtering* for recommendation in a customer-product graph. Given a product v purchased by a set of customer $U = \Gamma_v$ where Γ means the directed neighbors, we find another product v' and $U' = \Gamma_{v'}$. We recommend v' to the customers with purchase of v if $s_{v,v'} = |\Gamma_v \cap \Gamma_{v'}| / \min(|\Gamma_v|, |\Gamma_{v'}|) > \eta$, where η is a threshold and $s_{v,v'}$ is called the similarity of the two products. Thus, collaborative filtering can be based on BFS.
- *Local search refinement/reranking* in multimedia constructs a graph to represent a set of images [6] based on their similarities. The similarity can be measured by the scale invariant feature transform (SIFT). Given an image v, a heuristic method is utilized to map v to a vertex in the graph, say v'. Then, starting from v', we perform a local random walk with restart (RWR) to find a better match v''. The RWR allows a particle to walk from v' to $u \in \Gamma_{v'}$ at probability $p = \frac{1}{Z} s_{v,u}$. where $s_{\cdot,\cdot}$ is the similarity between two images and $Z = s_{v,v'} + \sum_{u \in \Gamma_{v'}} s_{v,u}$ is the normalizer.

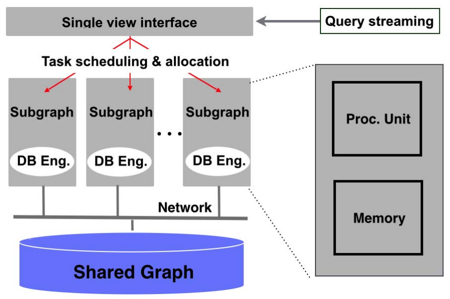


Figure 1: A sample shared-disk architecture

It is worth noting that in a real system there is typically a set of tasks involving subgraph traversal to be processed, and the tasks keep on streaming in. For the sake of improving system throughput, these tasks should be processed *concurrently*, using the parallel processing units in the platform.

Shared nothing (SN) architecture is prevalent in many data warehousing and database spaces to handle concurrency. Standard SN is a distributed computing architecture in which each node is independent and self-sufficient. However, since SN certainly takes longer to respond to queries that involve joins over large data sets from different partitions, many so-called SN systems are actually of the *shared disk* architecture [5]. Such architecture is illustrated in Figure 1. In our context, a big property graph is stored in a shared disk across a set of parallel compute nodes, each having its own CPUs and memory. A subgraph is loaded into a compute node if it is required by a task allocated to this node. The data access to local memory is much more efficient in terms of latency than that to the shared disk.

The architectural characteristics of the above enterprise big data processing platforms motivate us to explore scheduling scheme for achieving high throughput in concurrent graph processing. First, the subgraph traversal tasks must be allocated to the computing units evenly, so that the workload is *balanced* across the system. Second, due to the noticeable overhead in accessing data on the shared disk, data locality plays a critical role in graph traversal performance. If two subgraphs are largely overlapped, the corresponding travesal task should be allocated to the same processing unit. This is called *affinity* in this paper. The optimal scheduling scheme should maximize both the affinity and the workload balance. Despite the optimization towards general workload balance or task affinity has been extensively studied, it is non-trivial to combine them together appropriately for modeling graph traversal problems.

III. Related Work

There is a large body of literature on large scale graph processing from various disciplines, ranging from the big data analytic to high performance computing. Among various graph technologies, subgraph query and processing become a critical component in several big data scenarios. For example, Sun *et. al.* studied efficient subgraph matching in graphs with a billion vertices [7], based on the pipe-line join processing strategy and query optimization. Although the proposed method was parallelized, the resource allocation and workload balance were not discussed. Shao *et. al.* discussed subgraph listing in [8], but this work addressed a different area. Instead of exploring various subgraphs from a massive graph, it finds the occurrence of a particular subgraph. This is similar to subgraph indexing and matching discussed in [9]. Simmhan *et. al.* presented a framework called GoFFish [1], which is basically a subgraph-centric framework for large scale graph processing. It advances the traditional vertex-centric graph processing framework by processing each connected subgraph using shared memory multithreading model; while enable message passing across subgraphs. However, this framework assumes subgraphs are disjoint, but in reality many concurrent queries visit partially overlapped subgraphs, which results in affinity among subgraph traversals. In addition, there are several graph computing frameworks or packages which partially provide functionality on handling subgraph traversals [2][4][10][11][12][13]. However, to the best of our knowledge, none offers particular solution for addressing both the workload balance and subgraph affinity simultaneously on real enterprise big data platforms.

Parallel graph runtime and scheduling techniques from the perspective of HPC have been discussed in several publications, such as [14]. In this area, subgraphs are studied on distributed memory platforms [15], massively multithreading architectures [16], and GPGPUs [17], where the computations are often tuned for specific algorithms and architectures [15]. For example, the Parallel Graph Library (PGL) [18] offers diverse graph data abstractions, and can express level-synchronous vertex-centric BSP and coarse-grained algorithms over subgraphs, but the scheduling of activities on these subgraphs has not been well discussed yet. In addition, many of those techniques were discussed in the context of OpenMP and MPI, which are different from many enterprise level large data processing platforms nowadays in industry.

As mentioned in Section II, shared-nothing(SN) and shared-diskarchitectures are widely used in many enterprise level big data platforms for online transaction/analytic processing (OLTP/OLAP). However, such an architecture imposes severe challenges to graph computing, because of the irregular data access patterns. Therefore, publications of graph computing on SN architecture are quite limited. Muntés-Mulero *et. al.* discussed a graph partitioning technique for whole graph traversal (i.e., BFS) using SN [19]. A top-K aggregation over a large graph using SN is discussed in [20]. Another work related to graph on SN architecture is TriAD, a distributed RDF graph engine [21]. However, none of the work addresses techniques on dynamic scheduling schemes for subgraph traversals, no mention the tradeoff between workload balance and task affinity.

We utilize a variant auction algorithm for subgraph scheduling. Auction algorithm is essentially an optimization method [22], typically offering rich parallel activities. Early efforts on parallelizing auction algorithms assumed a shared repository of all task prices [23]. Zavlanos *et al.* proposed a MATLAB implementation of auction algorithms using nearest neighbor agreement protocols [24]. Due to the communication protocol and the overhead of MATLAB, the implementation shows limited scalability. Sathe *et al.* developed a parallel version of the auction algorithm on Cray XE6, achieving encouraging performance for some matrices [25]. However, their algorithm maintains a global price list, which the Cray XE6 architecture can support efficiently, but not a typical shared-nothing or shared-disk architectures. Riedy and Demmel [26] seem to have the first distributed implementation of the auction algorithm.

IV. Concurrent Subgraph Traversal

A. *Affinity in Subgraph Traversal*

In a throughput-oriented system, a set of query requests may arrive simultaneously. Each request provides a vertex, which is used as the starting vertex of a local subgraph traversal. As mentioned in Section II, by local traversal, we mean the query visits vertices and edges within the neighborhoods, say a few hops away from the starting vertex. Note that in this paper the traversal can behave as a BFS, but not necessarily visiting all the vertices, due to some user-defined predicates on the vertex/edge properties or attributes.

Consider two queries with starting vertices close to each other, such as the queries A and B shown in Figure 2. When the two corresponding traversals are performed, there are quite a few vertices being visited by both traversals. Therefore, in terms of data locality, if query A has been allocated to a processor, say p, we want to assign query B to p as well. This is called the affinity between queries A and B.

However, it is not convenient to calculate the mutual affinity between all pairs of queries because of following reasons: (1) Such calculation will result in $O(N^2)$ operations, given N queries. This can be time consuming when N is high. (2) The mutual affinity of all pairs does not explicitly give us a clear way to schedule queries onto processors. It may require additional clustering to form an allocation scheme.

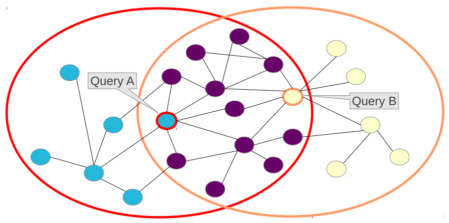
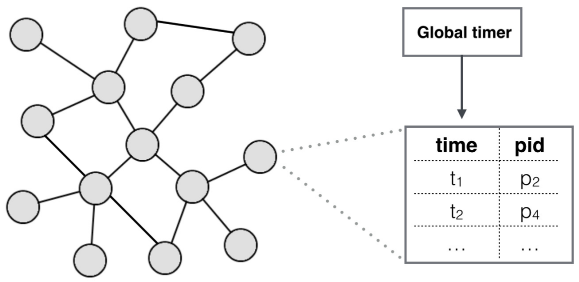


Figure 2: Subgraph affinity w.r.t. traversal



time	pid
t_1	p_2
t_2	p_4
…	…

Figure 3: Score affinity between a subgraph traversal and a processing unit using the list of recent visit signatures.

(3) The concurrent queries may keep on streaming in. The above method is of limited capability to handle the dynamic queries. Therefore, in this paper, instead of calculating the affinity between every two subgraph traversals, we compute the affinity between a subgraph traversal and a local processing unit (i.e. a processor plus its local memory).

We claim that a subgraph traversal is *affinitive* to a processor if some vertices in the subgraph have been recently visited by the processor. More specifically, we track a graph vertex v's recent visitor history using a list of signatures $L(v)$ shown in Figure 3. In this design, there is a global steady timer generating reference time stamps. Once a vertex is being assigned to a processor, we update the signature list by inserting a new pair of time stamp t_i from the global timer and the ID of the processor denoted by p_j, i.e., $L(v) \leftarrow L(v) \cup (t_i, p_j)$. Since we are only concerned about the recent visits, the list can be kept short, say 10 entries per vertex. Note that $L(v)$ is naturally ordered by time stamps. Thus, by browsing the list, we know which processors recently touched V:

We formulate the subgraph-processor affinity scoring function in two steps:

First, we model the affinity by ignoring the impact of the visit time. To quantify the affinity between a subgraph traversal and a processor, we use $s'_{v\to p}$ to denote the affinity score of allocating a subgraph traversal starting from $v \in G$ onto a processor $p \in \{0, 1, \cdots, P-1\}$, which considers the impact of data locality and caching characteristics:

$$s'_{v\to p} = \frac{\delta_{v,p} + \sum_{u\in\Gamma_v} \delta_{u,p}}{1 + |\Gamma_v|} \tag{1}$$

where $\delta_{v,p}$ is a variant Kronecker delta function that returns 1 if $p \in L(v)$ is satisfied; otherwise, it returns 0. Γ_v is the neighborhood of v. A larger neighborhood may further improve the accuracy of scoring; however, in such a case the scheduling itself results in a local traversal, although there is no heavy graph property or attributes involved. Thus, empirically, we only use the immediate neighbors of v for Γ_v and we found it performs well.

Then, let's impose the impact of time. Note that the affinity is also governed by time. The cached data tend to become expired after amount of time. We model this impact using a shifted negative exponential function. The function returns a coefficient between 0 and 1, which amends the score calculated in Eq. 1. Therefore, the subgraph-processor affinity score is estimated by:

$$s_{v\to p} = \left(e^{-\alpha(t-t_p)}\right) \cdot s'_{v\to p} \tag{2}$$

where t is read from the global timer at the evaluation time; t_p is the latest time stamp when v was visited by processor p. Parameter α controls the speed of the expiration:

$$\alpha = \frac{(n_p + n'_{t,t_p}) \cdot m}{M} \tag{3}$$

where n_p is the number of subgraphs allocated to processor p, but not yet executed; n'_{t,t_p} is the number of subgraphs traversed by processor p since time t_p; m is the average memory footprint taken by storing a subgraph and M is the total size of the memory space available to processor p. The rationale behind α is that, when the memory is saturated, a newly loaded subgraph tends to swap out the earlier cached subgraphs.

B. Affinity-based Scheduling

We allocate subgraph traversal tasks onto processors based on the affinity scoring function defined in Eq. 2. In order to illustrate the allocation method, we use a weighted bipartite graph $B(\mathcal{G}, \mathcal{P}, E, W)$ shown in Figure 4 to represent the relationship between subgraphs and processors, where $\mathcal{G}$, $\mathcal{P}$, E, W are the subgraph set, processing unit set, edges, and the weights on edges, respectively.

Basically, there is a pool of concurrently arrived traversal tasks to be assigned, which forms a set of vertices in the bipartite graph B. The other set of vertices in B consists of the available processing units, each with a task queue. For $G \in \mathcal{G}$ and $p \in \mathcal{P}$, we have edge $(G, p) \in E$ if and only if $W_{G,p} = s_{v\to p} > \eta$, where $v \in G$ is the starting vertex for traversing G; $s_{v\to p}$ is from Eq. 2 and η is a threshold to skip low affinity scores. Edge (G, p) means G and p are affinitive, so we can consider allocate G to p. The allocation is iterative. In each iteration, a processor will be assigned at most one subgraph, but we want to maximize the overall affinity.

Therefore, the allocation can be defined as selecting a subset of edges $E' \subseteq E$ from the bipartite graph B,

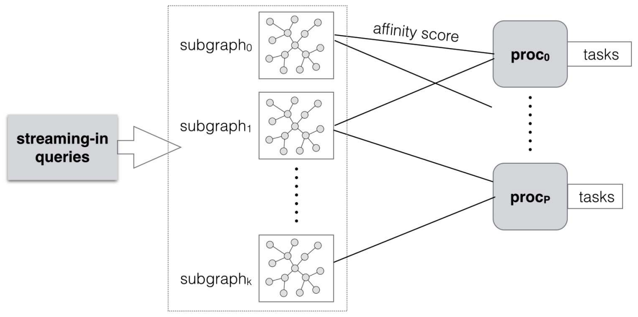


Figure 4: Affinity and balance aware subgraph scheduling is modeled by a dynamic weighted bipartite graph.

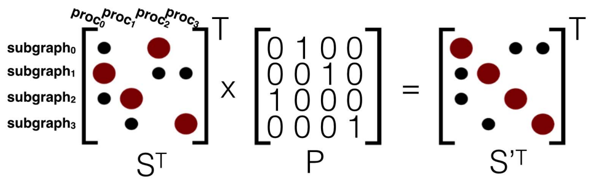


Figure 5: Model affinity-based subgraph scheduling as a matrix permutation

so that no two edges incline to the same vertex. In the meanwhile, the sum of the scores on the selected edges must be maximized. E' gives the allocation.

To formulate the above optimization, we represent bipartite graph B as a matrix $S = \{s_{v\to p}\}$, where each row represents a subgraph and each column represents a processor. The entry is the corresponding affinity. Selecting a subset of edges with maximum overall affinity scores is equivalent to find a permutation matrix P, so that it maximizes the sum of the diagonal of the permuted matrix. The sum of the diagonal is called the *trace* of the matrix, denoted by $tr()$. This formulation is shown in Figure 5.

C. Incorporation with Workload Balance

Each processing unit may have a queue of subgraph traversal tasks, as shown in Figure 3. To further improve performance, we want to keep the workload approximately balanced across the processors. This is especially important for real social networks where some vertices incline to a large number of edges. Thus, given a starting vertex v, even if a subgraph traversal G_v is affinitive to a heavily-loaded processor p, that is, $s_{v\to p}$ is high, we may want to allocate G_v to an idle processor p', if any.

We incorporate the workload of processors by weighting the affinity scoring matrix $S = \{s_{v\to p}\}$, $\forall v, p$. Let w_p denote the workload of processor $p \in \{0, 1, \cdots, P-1\}$, measured by the number of subgraphs in the task queue. We construct a reciprocal weight vector $\vec{w} = (\frac{1}{w_i+\tilde{\epsilon}})_{i=0,\cdots,P-1}$, where $\tilde{\epsilon}$ is a small positive number. We modify the S by:

$$A = S \odot (\vec{w}^T \vec{1}) = \left\{ \frac{s_{v\to p}}{w_p + \tilde{\epsilon}} \right\}_{\forall v,p} \tag{4}$$

where $\odot$ is the matrix inner product operator and $\vec{1} = (1, 1, \cdots, 1)^T$. We refer A as the *workload-aware affinity matrix*.

Based on the above discussion, We convert the above formulation into a linear programming as follows:

$$\begin{aligned} \max_{P} \quad & \text{tr}(A^T P) \\ \text{s.t.} \quad & P\vec{1}^T = \vec{1},\ P^T\vec{1} = \vec{1},\ P \geq 0, \end{aligned} \tag{5}$$

According to the linear programming [27], the above constraints ensures P is a permutation matrix. By solving this linear programming, P gives the scheduling scheme that allocates subgraphs to processors.

V. Parallel Incremental Auction

A. Dual Optimization

Efficient subgraph traversal scheduling requires a fast solution to Eq. 5. We solve the linear programming by converting it into an auction problem. Among various solutions to linear programming, the auction based approach usually offers the richest parallelism.

A constraint linear programming problem in form of Eq. 5 can be converted into a dual form. The auction algorithm [22] for solving the maximum weight bipartite matching problems solves the following dual form of the primal linear program in Equation 5:

$$\begin{aligned} \min_{p,\pi} \quad & \vec{1}^T p + \vec{1}^T \pi \\ \text{s.t.} \quad & \vec{1} p^T + \pi \vec{1}^T \geq A, \end{aligned} \tag{6}$$

where $p = (p_1, p_2, \cdots, p_N)^T$ and $\pi = (\pi_1, \pi_2, \cdots, \pi_N)^T$ are the dual vectors.

B. Auction Process

The auction algorithm computes the dual variables through an iterative *auction* process. Specifically, we view each row i as a buyer and each column j as an object; a_{ij} is the benefit of object j to buyer i; p_j is the price of object j (initialized to 0 at the very beginning); π_i is the profit for buyer i, defined as $(a_{ij} - p_j)$, in case object j is assigned to i. The auction algorithm proceeds as follows: Each buyer i bids for an adjacent object j that offers the highest profit, i.e., $\max_{j\in\text{adj}(i)}(a_{ij} - p_j)$. The bid is the difference between the highest profit and the second highest profit. An object j is assigned to the buyer offering the highest bid, say buyer i. Thus, row i and column j are matched and price p_j is increased by the bid. Price increment makes an object expensive for competitors, so that they can choose other objects. The above steps are repeated for all unmatched rows until every row is matched or the matching does not change any more.

The naive auction algorithm described above has a well known defect; i.e., it can stagnate due to a *price war* [22]. When the highest and second highest profits are equal for some buyer, the bid becomes 0 and the object price remains unchanged. Therefore, two buyers can infinitely compete for the same object without increasing its price. The object will be alternately assigned to them and the auction will not progress. To overcome this problem, a small positive scalar ϵ is added to the price of an object once it is assigned to a buyer.

The auction steps for subgraph scheduling are shown in Algorithm 1. In each iteration, the subgraphs (i.e. rows in Q) are processed. Lines 3–10 are the *parallel* auction steps (**for...pardo**), in which each unallocated subgraph traversal task i traverses its affinitive processors (i.e., the adjacent columns in adj(i)) simultaneously to find the best processor j_1 (the one offering the maximum profit $a_{ij} - p_j$) and the second best processor j_2. The price vector $\vec{p}$ is updated accordingly in Line 9. Note that the new price $a_{ij_1} - a_{ij_2} + p_{j_2} + \epsilon$ is the sum of the old price, ϵ, and the difference between the highest and second highest profits, $p_{j_1} + ((a_{ij_1} - p_{j_1}) - (a_{ij_2} - p_{j_2}))$, where $(a_{ij} - p_j)$ is the profit for i when choosing j. If the chosen processor was already allocated to another subgraph, then Lines 6–8 remove it from the matching result, i.e. the allocation M, and put the previously matched subgraph i' into Q'. The auction algorithm terminates when M is unchanged (Line 12).

Algorithm 1 SUBGRAPH_AUCTION

```
Input: Processing units $P = \{p_j\}_{j=0,\cdots,P-1}$; subgraph
    queries $Q = \{q_0, q_1, \cdots, q_k\}$, $k \leq |P|$; Workload-aware
    affinity matrix $A = \{a_{ij}\}$; minimum price increment $\epsilon$
Output: allocation scheme $M = \{(q_i, p_j)\}$
 1: repeat
 2:     $Q' \leftarrow \emptyset$
 3:     for subgraph $i \in Q$ pardo
            {The best & 2nd best columns}
 4:         $j_1 \leftarrow \arg\max(a_{ij} - p_j),\ \forall j \in$ adj($i$)
 5:         $j_2 \leftarrow \arg\max(a_{ij} - p_j),\ \forall j \in$ adj($i$), $j \neq j_1$
            {Update matching and price}
 6:         if $\exists i'$ s.t. $(i', j_1) \in M$ then
 7:             $M \leftarrow M \backslash \{(i', j_1)\}$, $Q' \leftarrow Q' \cup \{i'\}$
 8:         end if
 9:         $p_{j_1} \leftarrow a_{ij_1} - a_{ij_2} + p_{j_2} + \epsilon$, $M \leftarrow M \cup \{(i, j_1)\}$
10:     end for
11:     $Q \leftarrow Q'$, $W' = W$,$W = \sum_{(i,j)\in M} a_{ij}$
12: until $|W - W'| < \eta$
```

Note that the auction steps in Algorithm 1 will assign at most one subgraph to a processor. Thus, when the number of concurrent subgraph traversal requests is high, we will need to partition them into some segments, each consisting at most P subgraph traversal tasks, where P is the number of available processors. We must perform the above auction for each segment.

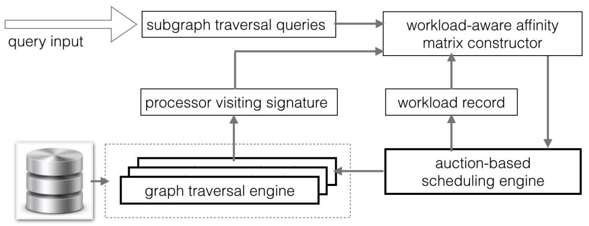

query input
subgraph traversal queries
workload-aware affinity matrix constructor
processor visiting signature
workload record
graph traversal engine
auction-based scheduling engine

Figure 6: Complete flow of workload-aware affinity scheduling of concurrent subgraph traversals

C. Subgraph Traversal

Once a subgraph traversal task is allocated to a processor, it can be performed locally. In our scenario, we consider a task q consisting of a starting a vertex v, a upper bound of local traversal depth h, some predicates/constraints θ to match with the edge property or vertex property during traversal. The subgraph traversal engine locates v and finds its neighbors, and so on. During the traversal, if any vertex/edge and/or its property is not cached in the memory yet, it must be loaded immediately. The loading of properties and checking them against θ can be much more time consuming compared to the traversal of the local graph structure. This is what we are particularly concerned about in many real industry solutions

D. Complete Workflow

The complete workflow is shown in Figure 6. The whole system runs as a service, receiving *subgraph traversal query* tasks, which are streaming in all the time. According to the number of available processors, it fetches the equal number of tasks, if any, from the input stream. These tasks are then fed into a component called *workload-aware affinity matrix constructor*. The affinity matrix is constructed based on the vertex *signatures* and the current *workload* of processors, as shown in Eq. 4. The matrix is consumed by the *auction-based scheduler* to compute the allocation of the resources based on Algorithm 1. According to the allocation scheme, the tasks are executed by a set of *subgraph traversal engines* running on a shared-disk scalable architecture.

VI. EXPERIMENTS

We implemented the proposed solution for property subgraph traversal using C++ on top of the IBM System G Native Store graph database [28] on a system with shared-disk and partitioned processing units (CPUs and memories). The code was compiled using g++ 4.7 with -O3 optimization in RHEL Linux CentOS 6.3. The scheduler is running on

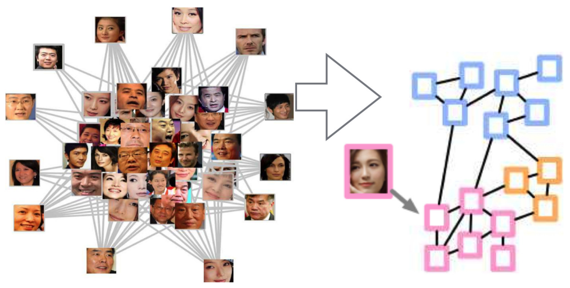


Figure 7: ISVision human face image graph is built based on the face SIFT similarities. It is partitioned into subgraphs.

an IBM BladeCenter with 4 Intel Xeon E7-4830 processors running at 2.13 GHz and the memory size is 256 GB. The scheduler and the property graph traversal engines communicate through a set of sockets.

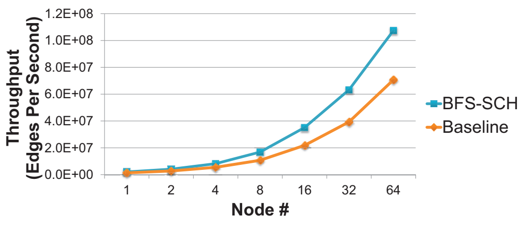
`Experiment Datasets` in our evaluations consist of: (1) A real twitter interaction graph created from GNIP [29] with 11,316,811 vertices and 85,331,846 edges, where each vertex represents an user and each edge represents the friendship/followership. The time stamp of retweets between two vertices and user information are given as the edge and vertex properties respectively. (2) A *real-world image reservoir* provided by the ISVision Co. for the ICME'2014 Industry Grand Challenge [30] (see Figure 7), including 5978 photos and 24 videos from 336 persons. The corresponding graph has 5978 vertices, 89,206 edges, and 45 partitions. An additional set of 1024 testing images was taken from those persons for generating queries. The graph of the face images were constructed according to the scale-invariant feature transform (SIFT) similarity, a widely used metric in multimedia. Thus, each vertex is an image and the edges connect similar images. The graph is clustered in preprocessing. When a query image comes, it is mapped to some clusters and then invokes a local search for search refinement. (3) A synthesized random graph with the same vertex and edge numbers as previous twitter graph. The property on the random graph conforms with that on the twitter interaction graph. We take this random graph into consideration because the twitter graph shows strong power law characteristics, resulting in highly skewed distribution of vertex degrees; while the random graph is much more balanced.

`Implementation Methodology` has been discussed in Section II. Basically, we evaluated three applications: For the twitter graph and the random graph, (1) we investigated the interaction within neighborhoods by performing BFS traversals from given vertices; (2) we searched shortest path between a pair of vertices (SSSP). For the image graph, (3) we performed a local re-ranking on the image reservoir for refining an image search. All the above applications involve local subgraph traversals.

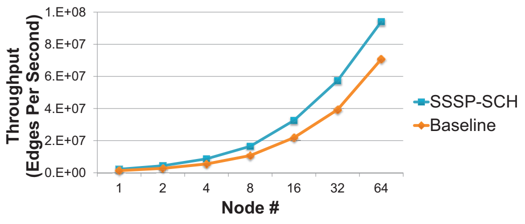
`Baseline system` in our experiments was following a random scheduling method with a first-come-first-serve (FCFS) based policy. In the baseline system, incoming queries were allocated to a randomly selected free unit. If all units were busy, an incoming query was then inserted into an arbitrary unit's query queue. In addition, each unit's queued queries were processed according to the FCFS order. In general, incoming queries were distributed to multiple units with a random scheduling method and FCFS processing order was followed within each unit.

`Throughput evaluation` was performed through the above three applications: BFS, SSSP, and image search, using various number of partitioned processors, ranging from 1 to 64. When a single processor is used, there is no workload balance issue. For each application we illustrate in Figure 8 the throughput of a baseline method and that of our proposed scheduling method, noted as *baseline* and *SCH*. The experimental results show consistent scalability of throughput as the number of processing units increases. The deployment of our proposed method doesn't compromise the existing scalability of baseline system. This is because the proposed scheduling method evenly distributes the workload with well considerations of workload distribution and balance. Due to the balanced workload and the affinity-based allocation, the performance is improved obviously compared to the *baseline* methods. The throughput speedup can achieve as high as 1.6x, 1.5x, 2.1x over the baseline system for BFS, SSSP, and image search, respectively. From the results, we can see that our proposed method archives both great throughput and scalability.

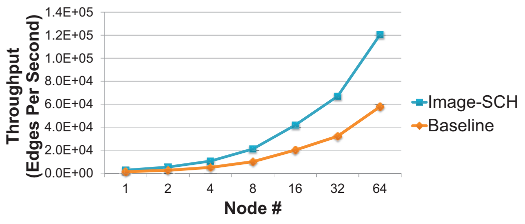
`Memory capacity sensitivity` was considered because the maximum allowed *reuse distance* of buffer data largely relies on memory capacity. A small memory leads to higher probability of swapping out history data and results in more access to the shared disk. It can bring significant performance loss. However, due to the increased latency, the pressure to scheduling efficiency is relaxed. So, we were able to run the auction based scheduling with smaller ϵ, which leads to improved scheduling scheme. We utilized 64 processing units and various sizes of memories, including 4GB, 8GB, 16GB, and unlimited. Note that although the physical memory is of a fixed size, IBM System G graph store allows us to configure the maximum memory footprint we want to use for buffering the graph elements. When the graph data is beyond the limit, earlier loaded graph data (subgraphs) is swapped out according to a LRU-like replacement policy. The results of memory capacity sensitivity are shown in Figure 9. From the results, we can see that throughput can be improved significantly with higher memory capacity. Meanwhile, compared with the baseline system, our proposed method shows much lower sensitivity of the increment in memory capacity. For example, by allowing unlimited memory capacity, the baseline system throughput can be improved by over 100% in BFS and SSSP. However, the



(a) BFS

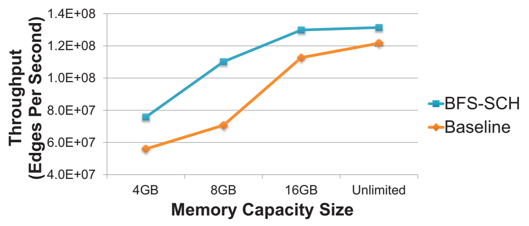


(b) SSSP

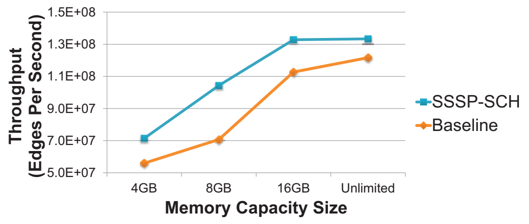


(c) Image search

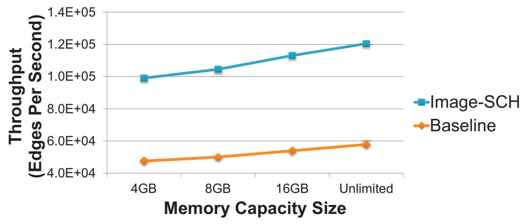
Figure 8: Throughput of BFS, SSSP, and image search w.r.t. number of partitioned processors/memories



(a) BFS



(b) SSSP



(c) Image Search

Figure 9: Illustration of memory capacity sensitivity on various applications

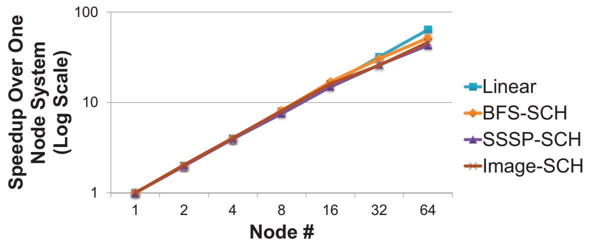
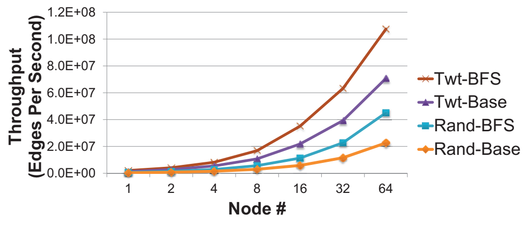


Figure 10: Speedup of concurrent subgraph traversal versus sequential subgraph traversal

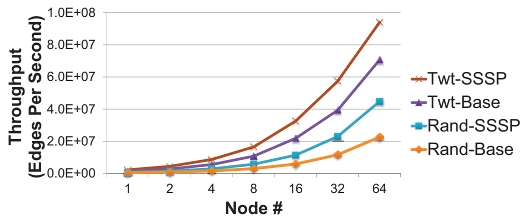
throughput of our proposed method achieves 80% of the maximum throughput with only 8GB memory capacity. This is in accordance our previous expectation. In our proposed method, by utilizing the data locality between queries, new queries have better chance to traverse mostly inside the in-memory buffer. Therefore, disk accesses as well as the buffer size requirements are reduced significantly. As shown, when the memory size reaches 16GB, our method achieves almost the same throughput as unlimited memory. So, the observation supports our expectation described above.

`Speedup over a single node` was investigated to demonstrate the scalability feature of our proposed method. It usually gives users a straightforward sense regarding the scalability of a parallel computing system. We selected the BFS application mentioned above because of its representativeness and performed the concurrent subgraph traversal with respect to different number of processors. The result is shown in Figure 10 with a logarithmic scale. The curve in the figure represents the speedup of the BFS application as the number of processing units increases. A linear curve is also included to represent the perfectly scalable case. Since the memory is partitioned, some vertices have to be loaded into multiple units; besides, the shared disk may also cause performance degradation when several processes access the on-disk data simultaneously. Therefore, the speedup is sublinear; however, as we can see from the figure, the speedup will keep increasing if we have more processing units available, which exhibits the excellent salability of the underlying system.

`Impact of input graph topology` was examined in our experiments to show the performance of the concurrent subgraph traversal applications with respect to a social graph with strong power law characteristic and a



(a) BFS



(b) SSSP

Figure 11: Impact of input graph topological types on its throughput of concurrent subgraph traversal

random graph with approximately even distribution of vertex degrees. The twitter interaction graph is a typical social graph where a few celebrity vertices have a large amount of inclined edges. We compared the performance using the twitter interaction graph and that using a synthesized random graph with evenly distribution of vertex degrees. The throughput of the two is shown in Figure 11. According to the experimental results, we can see that the throughput on the random graph is lower compared with the throughput on the twitter graph. This is caused by the topological difference between the two graphs. In the twitter graph, the local neighborhoods with high edge density make subgraph traversals have better chance to access visited vertices. In this case, it may traverse more edges when loading the same amount of vertices. Therefore, both baseline system and our proposed method show better throughput with twitter graph. In constrast, the random graph incorporate heavier requirements for vertex loading, which results in a much higher sensitivity of data affinity. Thus, the results in Figure 11 also show a better improvement over baseline system when using random graph.

`Performance improvement` of our proposed method is summarized in Figure 12. As shown, the throughput improvement of BFS can reach as high as 51.9% with 64 processors. Similarly, the SSSP application can also reach 50% improvement. Even in the worse case, 48% and 46% improvements were achieved respectively. The throughput improvement of image search application is much more significant. On average, a more than 2x throughput is shown by applying our proposed method. This is because of the a special feature of image graph: In the twitter graph, both vertex and edge properties are usually some small-sized meta data. However, the image graph attaches photo data to each vertex, leading to extremely large vertex properties. In this case, the performance penalty of loading vertices from disk storage is no longer reading small meta data, but loading large size photo data and also performing some image preprocessing. By applying our proposed method, the vertex loading operations were significantly reduced. Therefore, the reduced operations result in a huge throughput gain accordingly.

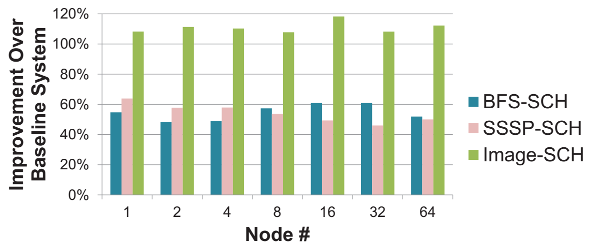


Figure 12: Improvement of proposed method over baseline system

VII. Conclusion

We addressed concurrent subgraph traversals in a large-scale property graph on industrial big data processing infrastructure for enterprise level operational analysis. The architectural characteristics of such infrastructure such as the shared-disk design lead to extremely high necessity on exploring the workload balance across processors with partitioned memory, and the task affinity based on data locality. Therefore, we proposed a new method to model the affinity and workload balance using a dynamic weighted bipartite graph. A variant auction algorithm is implemented to find optimal solutions for allocating a subgraph traversal task onto a processor. We parallelized the auction based scheduling method and conducted experiments on shared-nothing/shared-disk platforms. We showed the impact of various factors on the overall performance of concurrent subgraph traversals and illustrated the potentials of the proposed method on real enterprise level big data platforms using real datasets. In future, we would like to explore machine learning based approaches to optimizing the auction processing by finding an adaptive minimum price increment ϵ. We would also like to explore the distributed scheduling schemes for other enterprise level big data platforms.

References

[1] Y. Simmhan, A. G. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. S. Raghavendra, and V. K. Prasanna, "GoFFish: A sub-graph centric framework for large-scale graph analytics," *Lecture Notes in Computer Science*, vol. 8632, pp. 451–462, 2013.

[2] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases*. O'Reilly Media, Incorporated, 2013. [Online]. Available: http://books.google.com/books?id=RTvAmQEACAAJ

[3] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1006.4990*, 2010.

[4] "Titan distributed graph database," http://thinkaurelius.github.io/titan/, 2014.

[5] "IBM puredata for operatinoal analytics," www.ibm.com/software/data/puredata/analytics/, 2014.

[6] Y. Xia, J.-H. Lai, L. Nai, and C.-Y. Lin, "Concurrent image query using local random walk with restart on large scale graphs," in *IEEE International Workshop on Multimedia Big Data Computing*, 2014, pp. 1–6.

[7] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *Proc. VLDB Endow.*, vol. 5, no. 9, pp. 788–799, 2012.

[8] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu, "Parallel subgraph listing in a large-scale graph," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014, pp. 625–636.

[9] S. Zhang, J. Yang, and W. Jin, "SAPPER: Subgraph indexing and approximate matching in large graphs," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 1185–1194, 2010.

[10] "Tinkerpop," http://www.tinkerpop.com/, 2014.

[11] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proceedings of the 2013 international conference on Management of data*. ACM, 2013, pp. 505–516.

[12] "Apache giraph," https://giraph.apache.org/, 2014.

[13] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, vol. 8, 2012, pp. 31–46.

[14] D. Gregor and A. Lumsdaine, "The parallel bgl: A generic library for distributed graph computations," in *In Parallel Object-Oriented Scientific Computing (POOSC)*, 2005.

[15] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 65:1–65:12.

[16] D. Ediger and D. A. Bader, "Investigating graph algorithms in the bsp model on the cray xmt," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, 2013, pp. 1638–1645.

[17] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *Proceedings of the 14th International Conference on High Performance Computing*, 2007, pp. 197–208.

[18] H. A. Fidel, N. M. Amato, and L. Rauchwerger, "The stapl parallel graph library," in *Languages and Compilers for Parallel Computing*, vol. 7760, 2013, pp. 46–60.

[19] V. Muntés-Mulero, N. Martínez-Bazán, J.-L. Larriba-Pey, E. Pacitti, and P. Valduriez, "Graph partitioning strategies for efficient bfs in shared-nothing parallel systems," in *Proceedings of the 2010 International Conference on Web-age Information Management*, 2010, pp. 13–24.

[20] A. Chakraborty, "Top-k aggregation over a large graph using shared-nothing systems." in *BigData Conference*, 2013, pp. 448–457.

[21] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald, "TriAD: A distributed shared-nothing rdf engine based on asynchronous message passing," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014, pp. 289–300.

[22] D. Bertsekas, "The auction algorithm: A distributed relaxation method for the assignment problem," *Annals of Operations Research*, pp. 105–123, 1988.

[23] D. Bertsekas and D. Castañon, "Parallel synchronous and asynchronous implementations of the auction algorithm," *Parallel Comput.*, vol. 17, pp. 707–732, 1991.

[24] M. Zavlanos, L. Spesivtsev, and G. Pappas, "A distributed auction algorithm for the assignment problem," in *The 47th IEEE Conference on Decision and Control*, 2008, pp. 1212–1217.

[25] M. Sathe, O. Schenk, F. Müller, Y. Zhao, and B. Helmar, "Accelerating maximum weighted matching algorithms in massive graph analysis," *submitted*, 2011.

[26] E. J. Riedy and J. Demmel, "Sparse data structures for weighted bipartite matching," in *SIAM Parallel Processing for Scientific Computing*, 2004, pp. 1–2.

[27] D. Bertsekas, "Auction algorithms for network flow problems: A tutorial introduction," *Computational Optimization and Applications*, vol. 1, no. 1, pp. 7–66, 1992.

[28] "IBM System G," http://systemg.ibm.com/.

[29] "Gnip," https://gnip.com/, 2014.

[30] "ISVision Dataset," http://www.icme2014.org/isvision-challenge/.