GraphTwist: Fast Iterative Graph Computation with Two-tier Optimizations

Yang Zhou†, Ling Liu‡, Kisung Lee†, Qi Zhang‡
College of Computing, Georgia Institute of Technology
†{yzhou, kslee}@gatech.edu, ‡{lingliu, qzhang90}@cc.gatech.edu

ABSTRACT

Large-scale real-world graphs are known to have highly skewed vertex degree distribution and highly skewed edge weight distribution. Existing vertex-centric iterative graph computation models suffer from a number of serious problems: (1) poor performance of parallel execution due to inherent workload imbalance at vertex level; (2) inefficient CPU resource utilization due to short execution time for low-degree vertices compared to the cost of in-memory or on-disk vertex access; and (3) incapability of pruning insignificant vertices or edges to improve the computational performance. In this paper, we address the above technical challenges by designing and implementing a scalable, efficient, and provably correct two-tier graph parallel processing system, GraphTwist. At storage and access tier, GraphTwist maximizes parallel efficiency by employing three graph parallel abstractions for partitioning a big graph by slice, strip or dice based partitioning techniques. At computation tier, GraphTwist presents two utility-aware pruning strategies: slice pruning and cut pruning, to further improve the computational performance while preserving the computational utility defined by graph applications. Theoretic analysis is provided to quantitatively prove that iterative graph computations powered by utility-aware pruning techniques can achieve a very good approximation with bounds on the introduced error.

1. INTRODUCTION

Graph as an expressive data structure is popularly used to model structural relationship between objects in many application domains, such as social networks, web graphs, RDF graphs, sensor networks, protein interaction networks. These graphs typically consist of millions of vertices and billions of edges. Efficient iterative computation on such huge graphs is widely recognized as a challenging big data research problem, which has received heated attention recently. We can broadly classify existing research activities on scaling iterative graph computations into two categories: (1) Distributed solutions and (2) Single PC based solutions.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/. Obtain permission prior to any use beyond those covered by the license. Contact copyright holder by emailing info@vldb.org. Articles from this volume were invited to present their results at the 41st International Conference on Very Large Data Bases, August 31st - September 4th 2015, Kohala Coast, Hawaii.
Proceedings of the VLDB Endowment, Vol. 8, No. 11
Copyright 2015 VLDB Endowment 2150-8097/15/07.

Most of existing research efforts are dedicated to the distributed graph partitioning strategies that can effectively break large graphs into small, relatively independent parts [3, 13, 14, 23, 24, 27]. Several recent efforts [15, 20, 22, 25, 30] have successfully demonstrated huge opportunities for optimizing graph processing on a single PC through graph parallel abstractions that are efficient in both storage organization and in-memory computation.

However, to the best of our knowledge, existing approaches fail to address the following challenges:

- **Efficient vertex-oriented aggregation.** Existing graph abstractions mainly focus on parallelism strategies at each vertex and its associated edges (its adjacency list). However, the vertex-centric parallel tasks dramatically increase the inter-vertex communication overhead regardless whether the parallel tasks are executed in local memory or shared memory across a cluster of compute nodes. Thus, how to perform vertex-oriented aggregation operations at higher graph parallel abstractions to improve the efficiency remains to be very challenging for big graphs.
- **Efficient handling of large graphs with highly skewed vertex degree distribution and highly skewed edge weight distribution.** Large-scale graphs typically have skewed vertex degree distribution. Concretely, a relatively small number of vertices connect to a large fraction of graph, but a large number of vertices have very few or no links to other vertices. Some real-world graphs, say DBLP coauthor graph, have skewed edge weight distribution. Partitioning a large graph in terms of vertex partitions without considering skewed vertex degree distribution or edges with skewed weight distribution may result in substantial workload imbalance in parallel computation. We argue that different graph parallel abstractions should be supported at both access tier and computation tier to promote well-balanced computation workloads, better CPU resource utilization, and more efficient graph parallel executions.
- **Resource-adaptive partitioning of big graphs.** Given that different types of iterative graph applications combined with different sizes of graphs often have different resource demands on CPU, memory and disk I/O, choosing the right granularity of graph parallel abstractions allows big graph analysis to be performed on any commodity PC. An important technical challenge is to devise tunable multi-level graph parallel abstractions such that the graph processing system enables more balanced parallel workloads, more effective resource utilization for parallel computation, and at the same time offer the best access locality by maximizing sequential access and minimizing random access.

Graph	Type	#Vertices	#Edges	AvgDeg	MaxIn	MaxOut
Yahoo [29]	simple graph	1.4B	6.6B	4.7	7.6M	2.5K
Twitter [19]	simple graph	41.7M	1.5B	35.25	770.1K	3.0M
Facebook [12]	simple graph	5.2M	47.2M	18.04	1.1K	1.1K
DBLPS [1]	simple graph	1.3M	32.0M	40.67	1.7K	1.7K
DBLPM [1]	multigraph	0.96M	10.1M	21.12	1.0K	1.0K
Last.fm [2]	multigraph	2.5M	42.8M	34.23	33.2K	33.2K

Table 1: Real-world Datasets

- **Exploring graph utility-aware pruning techniques.** Iterative graph computations on large graphs with millions of vertices and billions of edges can generate the intermediate results that are orders of magnitude bigger than the original graph. One way to improve the computational performance on such big graphs is to prune those vertices or edges that do not directly contribute to the utility of graph computation as early as possible. For example, when computing the social influence of authors in the area of DB on the DBLP coauthor graph, those coauthor edges that are not DB specific can be pruned at early stage. Similarly, when computing the PageRank scores, those smaller elements in the transition matrix can be pruned in the subsequent iterations. Such pruning can significantly speed up the iterative graph computations while maintaining the desired graph utility. An important challenge is to design a graph storage data structure to flexibly support such utility-aware progressive pruning techniques.

To address these new challenges, we develop GraphTwist, an innovative graph parallel abstraction model with two-tier optimizations for processing big graphs with skewed vertex degree distribution and skewed edge weight distribution. The main contributions of this paper are summarized below.

- We present a scalable two-tier graph parallel aggregation framework, GraphTwist, for efficiently executing complex iterative computation tasks over large-scale graphs on a shared-memory multiprocessor computer.
- At storage and access tier, we propose a compact data structure for big graphs augmented with three divide-and-conquer graph parallel abstractions. We introduce the index structure, the basic algebra and its core operations to support resource-adaptive graph partitioning by selecting the right granularity of parallel abstractions based on both the system capacity and the utility of graph algorithm.
- At computation tier, we present two utility-aware pruning strategies: slice pruning and cut pruning, to further improve the performance by removing non-contributing vertices or edges while preserving the computational utility with bounds on the introduced error.
- Empirical evaluation over real large graphs demonstrates that GraphTwist outperforms existing representative graph parallel models in terms of both effectiveness and efficiency.

2. GRAPH PARALLEL ABSTRACTIONS

Large-scale graphs often have skewed vertex degree distribution. Table 1 shows the vertex degree distributions of several real graph datasets used in our experimental evaluation. For example, the Yahoo dataset has average vertex degree of 4.7 but maximum indegree of 7.6 million and maximum outdegree of 2.5 thousand. Similar observations can be made on several other datasets. Clearly, by replying on the simple vertex block based graph partitioning (i.e., each vertex and its adjacency list is in one partition), existing graph parallel models may result in very poor parallel performance due to substantial workload imbalance at vertex level in parallel computation. In addition, the processing time for vertices with small degree is very short compared to the in-memory and on-disk access latency, leading to inefficient CPU utilization in addition to poor workload balance.

Although many graph datsets (Yahoo, Twitter and Facebook) originally have 0/1 edge weights or do not have explicit edge weights, some real graphs (DBLP) have skewed edge weight distribution. In addition, in many iterative graph applications, we need to transform the original graphs into the weighted graphs. A typical example is the transition matrix of graph, which is widely applied to many real applications, such as PageRank, graph clustering and graph classification. Consider PageRank as an example, we need to iteratively calculate the multiplication between the transition matrix and the ranking vector. However, the transition matrix often has skewed edge weight distribution and small weight values may contribute little to the overall quality of the PageRank ranking result. Thus, GraphTwist is designed to address these issues by introducing two-tier optimizations to speed up iterative graph computations. The low-tier optimization, built on top of a scalable graph parallel engine GraphLego [35], utilizes a compact and flexible data structure and enables access locality optimization (Section 2). We introduce a graph utility-aware filtering method as the high-tier optimization, while preserving the desired quality with provable error bound (see Section 3 for detail).

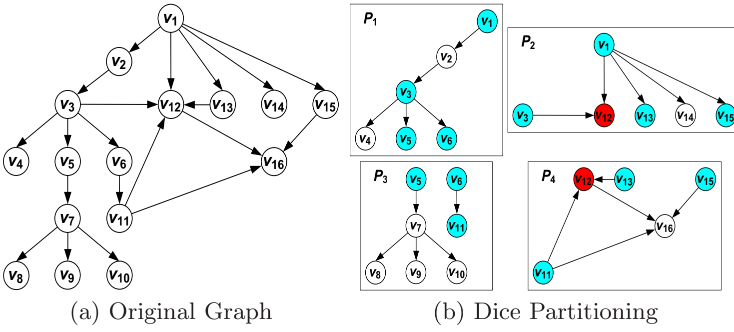
2.1 Modeling Graph as 3D Cube

In order to partition a large graph along different dimensions, we model a graph as a 3D cube, which allows us to devise three alternative graph parallel abstractions for partitioning a large graph: graph slice, graph strip, and graph dice. Each type of graph partitionings can be viewed as an alterative partition granularity and it partitions a graph into multiple subgraph blocks such that the edge sets of these subgraphs are disjoint and similar in size.

Let $G=(V,E,W)$ be a directed graph where V is a set of n vertices (i.e., $n=|V|$), E is a set of directed edges, and W is a set of weights of edges in E. Each vertex is associated with one or more states. Each edge $e=(u,v)\in E$ is referred to as the in-edge of destination vertex v and the out-edge of source vertex u respectively. Two vertices may be connected by multiple parallel edges. A graph G is modeled as a 3-dimensional representation of G, called **3D cube**, denoted as $I=(S,D,E,W)$ where $S=V$ represents the set of source vertices and $D=V$ specifies the set of destination vertices. If $u\in S$, $v\in D$ and $(u,v)\in E$, then $(u,v).weight=w\in W$ and (u, v, w) represents a cell with u, v, w as coordinates. We support three alternative types of graph partitioning: slice by dimension W, strip on slice by dimension S or D, and dice by all three dimensions.

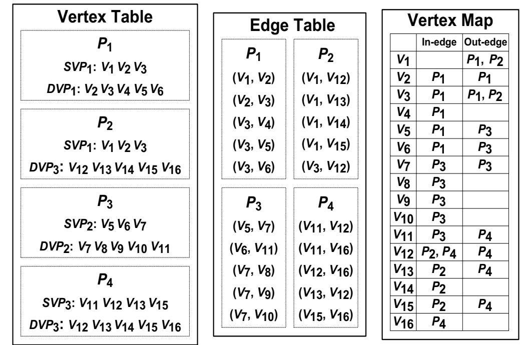
2.2 Dice Partitioning

The dice partitioning method partitions a large graph G into dices along all three dimensions of its 3D cube I and store G in dice subgraph blocks. Concretely, given $G=(V,E,W)$ and its 3D cube $I=(S,D,E,W)$, we first sort vertices in V by the lexical order of their vertex IDs. We then partition the destination vertices D into q disjoint partitions, called **destination-vertex partitions** (DVPs). Similarly, we partition the source vertices S into r disjoint partitions, called **source-vertex partitions** (SVPs). A **dice**



(a) Original Graph (b) Dice Partitioning

Figure 1: Dice Partitioning: An Example



	In-edge	Out-edge
V_1		P_1, P_2
V_2	P_1	P_1
V_3	P_1	P_1, P_2
V_4	P_1	
V_5	P_1	P_3
V_6	P_1	P_3
V_7	P_3	P_3
V_8	P_3	
V_9	P_3	
V_{10}	P_3	
V_{11}	P_3	P_4
V_{12}	P_2, P_4	P_4
V_{13}	P_2	P_4
V_{14}	P_2	
V_{15}	P_2	P_4
V_{16}	P_4	

Figure 2: Dice Partition Storage (IEDs and OEDs)

of I is a subgraph of G, denoted as $H{=}(S_H, D_H, E_H, W_H)$, satisfying the following conditions: $S_H{\subseteq}S$ is a SVP, specifying a subset of source vertices, $D_H{\subseteq}D$ is a DVP, denoting a subset of destination vertices, $W_H{\subseteq}W$ is a subset of edge weights, and $E_H{=}\{(u,v)|u{\in}S_H, v{\in}D_H, (u,v){\in}E, (u,v).weight$ $\in W_H\}$ is a set of directed edges, each with its source vertex from S_H, its destination vertex from D_H and its edge weight in W_H. We maintain two types of dices for each vertex v in G: one is **in-edge dice** (IED) containing only in-edges of v and another is **out-edge dice** (OED) containing only out-edges of v. Edges in each IED (OED) are stored by the lexical order of their source (destination) vertices. Unlike a vertex and its adjacency list (edges), a dice is a subgraph block comprising a SVP, a DVP and the set of edges that connect source vertices in the SVP to the destination vertices in the DVP. Thus, a high-degree vertex and its edges are typically partitioned into multiple dices with disjoint edges but overlapping vertices. Also some pairs of DVP and SVP may not form a dice partition when $E_H{=}\emptyset$. Figure 1 (b) shows a dice partitioning of an example graph in Figure 1 (a), consisting of four dices with disjoint edge sets and the vertices in multiple dices are highlighted.

To provide efficient support for different graph applications that use either or both of in-edges and out-edges, an original graph is stored with two types of 3D cubes: **in-edge cube** and **out-edge cube**, each consists of unordered dices with the same type on disk (IEDs or OEDs). Figure 2 shows the storage organization of the dice partitions in Figure 1 (b), consisting of a vertex table, an edge table, and the mapping of vertex ID to the list of in-edge partition IDs and the list of out-edge partition IDs.

2.3 Slice Partitioning

The slice partitioning method is an effective parallel abstraction to deal with the skewed edge weight distribution. It partitions a 3D cube of a graph G into p slices along dimension W. p is chosen such that edges with similar weights are clustered into the same partition. Let $I{=}(S, D, E, W)$ denote a 3D cube of graph $G = (V, E, W)$. We define a **slice** of I as $J{=}(S, D, E_J, W_J)$ where $W_J{\subseteq}W$ is a subset of edge weights, and $E_J{=}\{(u,v)|u{\in}S, v{\in}D, (u,v).weight{\in}W_J, (u,v)$ $\in E\}$ is a set of directed edges from S to D with weights in W_J. We maintain two kinds of slices for G: **in-edge slices** containing in-edges of destination vertices in D and **out-edge slices** comprising out-edges of source vertices in S.

Consider the DBLP coauthor multigraph as an example, it allows for parallel edges between a pair of coauthor vertices, each edge denoting the number of coauthored publications in one of 24 computer research fields [9]: AI, AIGO, ARC, BIO, CV, DB, DIST, DM, EDU, GRP, HCI, IR, ML, MUL, NLP, NW, OS, PL, RT, SC, SE, SEC, SIM, WWW. Figure 3 shows an illustrative example of a coauthor multigraph with three types of edges: AI, DB and DM, representing the number of coauthored publications in AI conferences (*IJCAI*, *AAAI* and *ECAI*), DB conferences (*SIGMOD*, *VLDB* and *ICDE*), and DM conferences (*KDD*, *ICDM* and *SDM*), respectively. By slice partitioning, we obtain three slices, one for each category in the bottom of Figure 3. Clearly, to compute the coauthor-based social influence among researchers in the area of DB, we only need to perform iterative computation with joint publications on DB conferences by using the DB slice in Figure 3.

In addition, we can speed up iterative graph algorithms on a simple graph (non-multigraph) by performing parallel computation on p slices.

2.4 Strip Partitioning

The strip partitioning method cuts a slice of the 3D cube of a graph G along either dimension S or dimension D to obtain out-edge or in-edge strips. Compared to the dice partitioning method that cuts the 3D cube of a graph (or a slice of graph) along both S and D, strips represent larger partition units than dices. A strip can be viewed as a sequence of dices stored physically together. By cutting an in-edge slice $J{=}(S, D, E_J, W_J)$ along dimension D into q in-edge strips, with q being the number of DVPs, each strip is denoted as $K{=}(S, D_K, E_K, W_J)$, where $D_K{\subseteq}D$ is a DVP, and $E_K{=}\{(u,v)|u{\in}S, v{\in}D_K, (u,v).weight{\in}W_J, (u,v){\in}E_J\}$ is a set of directed edges from S to D_K with weights in W_J. An **in-edge strip** contains all IEDs of a DVP. Similarly, an **out-edge strip** can be defined as all OEDs of a SVP. Figure 3 gives an example of strip partitioning over the DB slice: (1) all coauthored DB links of *Daniel M. Dias*, *Michail Vlachos* and *Philip S. Yu*, and (2) all coauthored DB links of *Jiawei Han*, *Sangkyum Kim* and *Tim Weninger*.

Vertex Cut. A vertex cut is introduced in GraphTwist as a logical partition unit. A dice partition can be viewed as a subgraph composed of multiple vertex cuts, one per vertex. Formally, given an IED $H{=}(S_H, D_H, E_H, W_H)$, an **out-edge cut** of H is denoted as $L(u){=}(u, D_H, E_L, W_H)$ where $u{\in}S_H$ and $E_L{=}\{(u,v)|v{\in}D_H, (u,v){\in}E_H, (u,v).weight{\in}W_H\}$. $L(u)$ contains all out-edges of u in this IED. Since edges in each IED are stored by the lexical order of their source vertices, we can easily split an IED into multiple out-edge cuts. Similarly, an OED can be viewed as a set of **in-edge cut**s, each containing the in-edges between a set of source

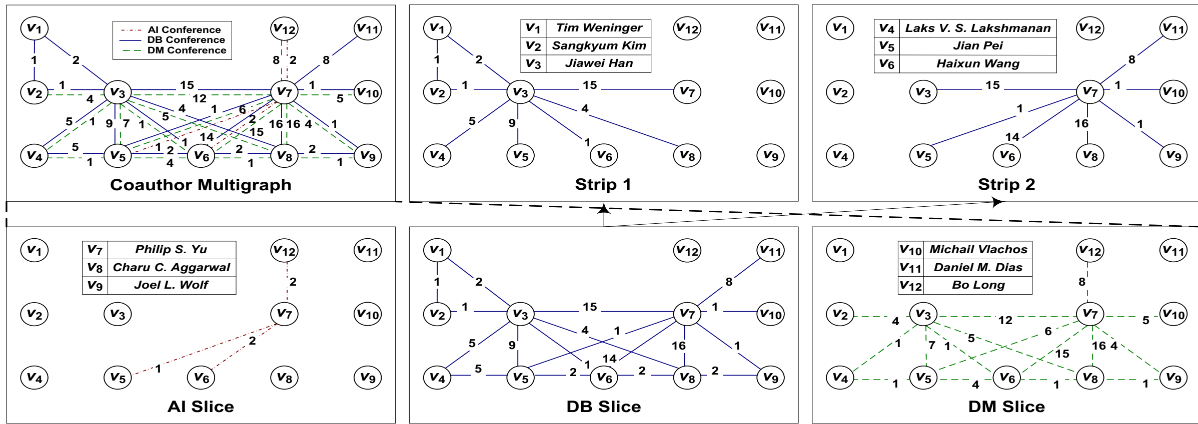


Figure 3: Slice Partitioning and Strip Partitioning: A Real Example Extracted from DBLP

vertices and a destination vertex. One of the utility-aware pruning optimization at the computation tier is vertex-based pruning based on vertex cuts (see Section 3.2 for detail).

2.5 Locality-aware Graph Partitioning

We use our API function *GraphLoad* to load an input graph into the GraphTwist store by building its 3D cube representation. We then invoke the *GraphPartition* routine to obtain the graph partitions at the chosen partition granularity (slice, strip or dice), and build the partition-level index and the vertex-to-partition index. The graph metadata is collected during the building time, such as vertex degree, directed or undirected graph, and so on.

By partitioning a big graph into multiple subgraphs based on the chosen partition granularity (slice, strip or dice), we divide iterative computation on big graph into iterative computations on multiple subgraphs in parallel with two steps: (1) calculate partial vertex updates in each subgraph in parallel; and (2) aggregate partial vertex updates from all subgraphs to generate complete vertex update.

2.6 Partition Index

To enable fast access to different kinds of partitions, a general index structure is used to index slices, strips or dices, depending on the choice of specific partitioning methods. The dice-level index is a dense index, where each index entry maps a dice ID and its SVP (or DVP) to the chunks on disk where the dice block is physically stored. The strip-level index is a two-level sparse index, where each index entry maps a strip ID and its DVP (or SVP) to the dice-level index entries relevant to this strip and then maps each dice ID and its SVP (or DVP) to the dice chunks in the physical storage. It enables fast access to dices with a strip-specific condition. Similarly, the slice-level index is a three-level sparse index with slice index entries at the top, strip index entries at the middle and dice index entries at the bottom, enabling fast retrieval of dices with a slice-specific condition. We also maintain a vertex-to-partition index that maps each vertex to the set of partitions containing in-edges or out-edges of this vertex, as shown in Figure 2.

2.7 Programmable API with Multi-threads

We provide a conventional vertex-oriented programming interface to enable users to write their iterative graph applications through a set of library functions. Users only need to define their iterative algorithms in terms of vertex-level computation using the library functions, such as *Scatter* and *Gather*. The *Scatter* function on a vertex v sends its vertex state from the previous iteration to a set of v's neighbors, say the destination vertices of the out-edges of v. The *Gather* function on a vertex v works on a set of v's neighbors, e.g., the source vertices of the in-edges of v, to aggregate their vertex states in order to update its vertex state. GraphTwist will compile the user-provided code as a sequence of GraphTwist internal function (routine) calls that understand the internal data structure for accessing the graph by subgraph blocks. These routines can carry out the iterative computations on each vertex in the input graph slice by slice, strip by strip, or dice by dice. For example, PageRank algorithm can be written by simply proving the computation tasks for a vertex, as shown in Algorithm 1.

2.8 Partition-level/Vertex-level Parallelism

GraphTwist supports parallel computation at two levels: (1) the partition level (slice, strip or dice) and (2) the vertex level. In GraphTwist, the edges in different partitions are disjoint no matter whether the partition unit is slice, strip or dice. In addition, the vertices in different in-edge (out-edge) dices or strips by design belong to different DVPs (SVPs), which are disjoint from each other (recall Figure 2), even if a vertex may belong to both a DVP and a SVP. Thus, we group the dice or strip partitions by their DVPs (SVPs) such that the partitions associated to the same DVP (or SVP) form a DVP (or SVP) specific partition group. The iterative computation on a graph is implemented as the iterative computation on each DVP (or SVP) specific partition group. This ensures that the vertex update can be safely executed on multiple strip or dice groups in parallel. Moreover, the vertex update can also be safely executed on multiple vertices within each strip or dice group in parallel since each vertex within the same DVP (or SVP) is unique. For slice partitions, each vertex within the same slice is unique. Thus, GraphTwist can safely perform the parallel computation at both the partition level and the vertex level.

2.9 Partial Vertex Update

However, when a DVP (or SVP) contains too many high-degree vertices, we may need to perform the parallel partial updates at both the partition level and the vertex level.

Algorithm 1 PageRank

```
Initialize(v)
  v.rank = 1.0;
Scatter(v)
  msg = v.rank/v.degree;
  //send msg to destination vertices of v's out-edges
Gather(v)
  state = 0;
  for each msg of v
  //receive msg from source vertices of v's in-edges
    state += msg; //summarize partial vertex updates
  v.rank = 0.15 + 0.85 * state; //produce complete vertex update
```

Figure 4 presents an example of partial update at the partition level. There are k available threads $T_1, \cdots, T_k$ in the system and m associated edge partitions $P_1, \cdots, P_m$ in the partition group of SVP_j. One thread processes only one edge partition at a time. We classify vertices in each partition into three categories: (1) internal vertices belonging to only one partition; (2) border vertices belonging to multiple partitions but their vertex updates are confined to only one partition by the given graph algorithm; and (3) critical border vertices belonging to multiple partitions and their vertex updates depends on multiple partitions. In the example of Figure 2, given the graph application of PageRank, which only uses in-edges, the partition group of DVP_1 contains only one dice partition P_1. v_2 and v_4 are internal vertices, and v_3, v_5 and v_6 are border vertices. Similarly, the partition group of DVP_2 contains only one dice partition P_3, v_7, v_8, v_9, and v_{10} are internal vertices, and v_{11} is border vertex. The partition group of DVP_3 contains two dice partitions: P_2 and P_4. Vertex v_{14} is internal vertex, and v_{13} and v_{15} are border vertices for P_2. Vertex v_{16} is internal vertex for P_4. Vertex v_{12} is the only critical border vertex for both P_2 and P_4, because only v_{12} receives partial updates from both partitions.

For internal vertices and border vertices, we can commit their vertex updates upon the completion of the partition-level parallel computation in each iteration. However, for critical border vertices (e.g. v_i in Figure 4), its associated edges may distribute into multiple edge partitions, say $P_1, \cdots, P_m$. In order to avoid conflict, GraphTwist maintains a partial update list with an initial *counter* of 0 for each critical border vertex in memory. If a thread T_x $(1 \leq x \leq k)$ processes an edge partition P_y $(1 \leq y \leq m)$ and needs to update the state of v_i, T_x first executes the *Scatter* process to put the partial update $v_{i,y}$ by P_y (a temporary local copy of vertex v_i) into the partial update list, and then check if *counter* is equal to the number of v_i's associated edge partitions from the vertex map, as shown in Figure 2. If not, T_x performs *counter*++ and the scheduler assigns an unprocessed edge partition to T_x. Otherwise, we know that P_y is the last processed edge partitions (e.g. P_m in Figure 4). Thus, T_x continues to perform the *Gather* process to aggregate all partial updates of v_i in its partial update list to generate a complete update of v_i.

2.10 Number of Partitions

Typically, we partition a large graph into p slices along dimension W with each slice of roughly the same size. The parameter p is determined based on the estimation of whether each slice and its intermediate computation results will fit into the available memory. If a slice and its intermediate results are too big to fit into the working memory for a

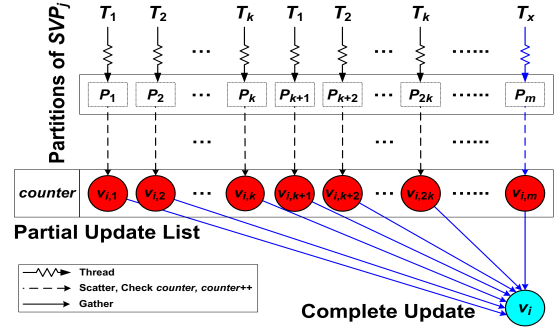


Figure 4: Multi-threading Update and Asynchronization

given graph application, we continue to partition the slice into q in-edge strips (or out-edge strips) along dimension D (or S). The setting of q should ensure that each in-edge strip (or out-edge strip) and its intermediate results will fit into the available memory. Upon exceptions such as skewed vertex degree distribution, we continue to partition the in-edge strips (or out-edge strips) into r partitions with equal size, resulting in $q \times r$ associated IEDs (or OEDs). r should be chosen such that each IED (or OED) can be fully loaded into the memory from disk.

However, such online simple estimation-based strategy for setting the parameters p, q and r may not be optimal. This motivates us to design an offline regression-based learning method. First, we model the nonlinear relationship between independent variables p, q or r and dependent variable T (the runtime) as an nth order polynomial by utilizing multiple polynomial regression [10]. We obtain a regression model that relates T to a function of p, q, r, and the undetermined coefficients α: $T \approx f(p, q, r, \alpha) = \sum_{i=1}^{n_p} \sum_{j=1}^{n_q} \sum_{k=1}^{n_r} \alpha_{ijk} p^i q^j r^k + \epsilon$ where n_p, n_q and n_r are the highest orders of variables p, q or r, α_{ijk} are the regression coefficients, and ϵ represents the error term of the model. We then select all possible m samples of (p_l, q_l, r_l, T_l) $(1 \leq l \leq m)$ from the existing experiment results, such as the points in Figure 16 (b), to generate m linear equations:

$$T_l = \sum_{i=1}^{n_p} \sum_{j=1}^{n_q} \sum_{k=1}^{n_r} \alpha_{ijk} p_l^i q_l^j r_l^k + \epsilon, \qquad 1 \leq l \leq m \tag{1}$$

We adopt the least squares approach [6] to solve the above overdetermined linear equations and generate α_{ijk} and ϵ. Finally, we utilize a successive convex approximation method (SCA) [16] to solve this polynomial programming problem with the objective of minimizing the predicted runtime, i.e., $\min_{p,q,r} \sum_{i=1}^{n_p} \sum_{j=1}^{n_q} \sum_{k=1}^{n_r} \alpha_{ijk} p^i q^j r^k + \epsilon \; s.t. \; 1 \leq p \leq |W|, 1 \leq q \leq n, 1 \leq r \leq n$, to generate the optimal p, q and r.

3. GRAPH UTILITY-AWARE PRUNING

Many large-scale real-world graphs have millions of vertices and billions of edges. Handling such big graphs in memory may require tens or hundreds of gigabytes of DRAM. Popular iterative graph applications usually consist of a series of matrix-vector computations [7,17,34] or matrix-matrix computations [18, 26, 31–33, 36]. Besides processing graphs partition by partition, another way to address this problem is to speed up the iterative computations by pruning some insignificant vertices or edges based on a certain statistical measure. For example, PageRank needs to iteratively calculate the multiplication between the transition matrix

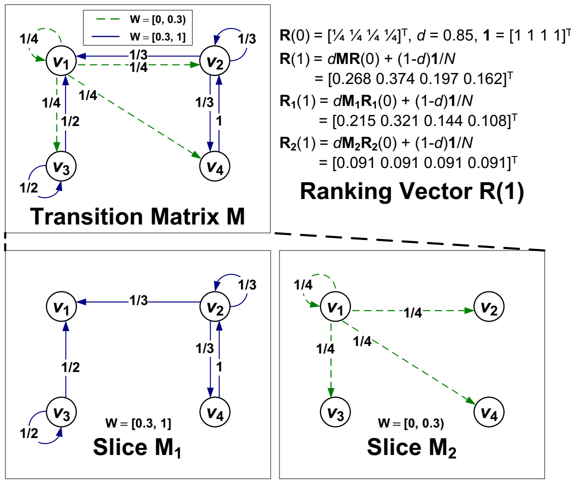


Figure 5: PageRank with Slice Pruning

$\mathbf{M}$ and the ranking vector $\mathbf{R}$. In GraphTwist, we set rank entries in $\mathbf{R}$ as vertex states and elements in $\mathbf{M}$ as edge weights. Given that the core utility of PageRank is to produce an ordered list of vertices by their PageRank scores, we want to prune some insignificant edges to speed up the PageRank computation while preserving the ranking order of all vertices. Figure 5 presents the transition matrix $\mathbf{M}$ of a given graph with 0/1 edge weights. According to the edge weight distribution in $\mathbf{M}$, we partition $\mathbf{M}$ into two slices: $\mathbf{M_1}$ with edge weights of $[0.3, 1]$ and $\mathbf{M_2}$ with edge weights of $(0, 0.3)$. Given an initial ranking vector $\mathbf{R}(0)$ and a damping factor d, we utilize the transition matrix $\mathbf{M}$ to calculate the exact ranking vector $\mathbf{R}(1)$ and use the selected slice $\mathbf{M_1}$ to compute the approximate ranking vector $\mathbf{R_1}(1)$ since the pruned edges in $\mathbf{M_2}$ have relatively small weights. Although the corresponding ranking scores in $\mathbf{R}(1)$ and $\mathbf{R_1}(1)$ are somewhat different, the ranking orders in the two ranking vectors are identical, indicating that the pruning of slice $\mathbf{M_2}$ preserves the utility of PageRank.

This motivates us to propose two utility-aware pruning techniques: slice pruning and cut pruning, to speed up the iterative graph computations while preserving the computational utility. Given that matrix multiplication is the fundamental core for many iterative graph applications, we use the matrix multiplication as an example to describe two pruning techniques: (1) speed up matrix multiplication by pruning insignificant slices with provable error bound, and (2) accelerate strip multiplication by pruning trivial vertex cuts in the strips with bounded error.

3.1 Slice Pruning (Subgraph-based Pruning)

The main idea is to reduce the computational cost by pruning some insignificant slices based on a statistical measure while preserving the utility of the graph computation. To speed up the iterative computations, one intuitive idea is to prune those sparse slices with small weights. We introduce the concept of slice density as a statistical measure to evaluate the importance of a slice. For presentation convenience, we use a symbol to represent a graph partition and its matrix (or vector) representation interchangeably when no confusion occurs. Cube, slice, strip and dice correspond to matrices, and cut corresponds to vector.

Definition 1. *[Slice Density] Let I=(S, D, E, W) be a 3D cube of a directed graph G=(V, E, W), and J=(S, D, E_J, W_J) be a slice of I where S=V, D=V, $W_J \subseteq W$, and E_J =$\{(u, v)|u\in S, v\in D, (u, v).weight\in W_J, (u, v)\in E\}$, the density of J is defined as follow.*

$$SliceDensity(J) = \|J\|_F = \sqrt{\sum_{u=1}^{n}\sum_{v=1}^{n}\sum_{(u,v)\in E_J} w(u, v)^2} \quad (2)$$

where J denotes the slice itself as well as its matrix representation, F is the Frobenius norm, and $w(u, v)$ represents the weight of an edge (u, v). If a slice is dense and has large edge weights, then it will have a large density.

Next we illustrate how to use the *SliceDensity* measure for matrix multiplication with slice pruning to prune the sparse slices with small weights. Given two matrices A consisting of s out-edge slices $A_1, \cdots, A_s$ and B comprising t in-edge slices $B_1, \cdots, B_t$, we decompose a matrix multiplication C=$A\times B$ into $s*t$ slice multiplications, i.e., C=$\sum_{k=1}^{s}\sum_{l=1}^{t} A_k \times B_l$. We propose a Monte-Carlo algorithm to compute a multiplication approximation: choose "important" $x*y$ ($<< s*t$) of $s*t$ slice multiplications to calculate the multiplication, while the expectation of the approximate multiplication is equal to the exact multiplication. Slice pruning strategy is two-fold: (1) pick slices according to the amount of "information" the slices contain; (2) rescale the multiplication to compensate for the slices that are not picked.

Concretely, we first compute the selection probability ρ_{kl} of each pair of slices (A_k and B_l) in terms of their precomputed *SliceDensity* values. Then we independently do $x*y$ selection trials and choose $x*y$ slice pairs from the original $s*t$ slice pairs in terms of selection probabilities. The indices of extracted slices of A and B in the hth trial are kept in $\Phi(h)$ and $\Psi(h)$ respectively. GraphTwist implements the slice multiplication at the strip level by invoking the **StripMultiply** method in Algorithm 3. We also rescale the edges in F with rescaling factor $\frac{1}{x*y*\rho_{\Phi(h)\Psi(h)}}$ to compensate for the slices that are not picked, where $\rho_{\Phi(h)\Psi(h)}$ denotes the selection probability of the $\Phi(h)^{th}$ slice of A and the $\Psi(h)^{th}$ slice of B. Thus, the multiplication approximation is calculated as $\tilde{C}$=$\sum_{h=1}^{x*y} \frac{A_{\Phi(h)}\times B_{\Psi(h)}}{x*y*\rho_{\Phi(h)\Psi(h)}}$. The edges can be safely updated in parallel at strip level since F (or E) corresponds to a unique IED and a unique OED of $\tilde{C}$ such that the IEDs

Algorithm 2 MatrixMultiply(A, B, x, y, q, z)

1: Initialize $E[1\cdots\lfloor n/q\rfloor][1\cdots\lfloor n/q\rfloor]$;
2: **for** $k = 1, \cdots, s$
3: **for** $l = 1, \cdots, t$
4: $\rho_{kl} = \frac{SliceDensity(A_k)*SliceDensity(B_l)}{\sum_{i=1}^{s}\sum_{j=1}^{t} SliceDensity(A_i)*SliceDensity(B_j)}$;
5: **for** $h = 1, \cdots, x*y$ independently
6: Pick $a, b, a \in \{1, \cdots, s\}, b \in \{1, \cdots, t\}$ with Prob$(a = k, b = l)$ $= \rho_{kl}, k = 1, \cdots, s, l = 1, \cdots, t$;
7: $\Phi = \Phi \cup \{a\}$; $\Psi = \Psi \cup \{b\}$;
8: **parallel for** $i = 1, \cdots, q$
9: **parallel for** $j = 1, \cdots, q$
10: **for** $h = 1, \cdots, x*y$
11: F = **StripMultiply**$(A, \Phi(h), i, B, \Psi(h), j, z)$;
12: **parallel for each** $e \in F$
13: $E[e.source][e.destination].AddWeight(e.weight/(x*y*\rho_{\Phi(h)\Psi(h)}))$;
14: Split E into p partitions in terms of weight distribution;
15: Write $IED[j][i]$ in p in-edge slices of $\tilde{C}$ to disk;
16: Write $OED[i][j]$ in p out-edge slices of $\tilde{C}$ to disk;

(or OEDs) generated by different strip multiplications in the same slice pair are irrelevant to each other.

The following theorems state that our slice pruning strategy can achieve a good approximation with bounded error.

THEOREM 1. *Given an actual multiplication* $C=\sum_{k=1}^{s}\sum_{l=1}^{t} A_k \times B_l$*, a multiplication approximation* $\tilde{C}=\sum_{h=1}^{x*y}\frac{A_{\Phi(h)}\times B_{\Psi(h)}}{x*y*\rho_{\Phi(h)\Psi(h)}}$*, then the expectation* $E(\tilde{C}(i,j))=C(i,j)$ *for* $\forall i,j\in\{1,\cdots,n\}$.

Proof. Suppose that a random variable $X_h=\frac{A_{\Phi(h)}\times B_{\Psi(h)}}{x*y*\rho_{\Phi(h)\Psi(h)}}(i,j)=\sum_{m=1}^{n}\frac{A_{\Phi(h)}(i,m)*B_{\Psi(h)}(m,j)}{x*y*\rho_{\Phi(h)\Psi(h)}}$ *where* $h\in\{1,\cdots,x*y\}$. X_h *represents the entry in the* i^{th} *row and* j^{th} *column of multiplication between slice* $A_{\Phi(h)}$ *and slice* $B_{\Psi(h)}$ *with edge rescaling of* $\frac{1}{x*y*\rho_{\Phi(h)\Psi(h)}}$*, then all of* $X_h s$ *are independent random variables. Also,* $\tilde{C}(i,j)=\sum_{h=1}^{x*y}X_h$*. Thus,* $E(\tilde{C}(i,j))=E(\sum_{h=1}^{x*y}X_h)=\sum_{h=1}^{x*y}E(X_h)$ *since all of* $X_h s$ *are defined on the same probability space. In addition,* $E(X_h)=\sum_{k=1}^{s}\sum_{l=1}^{t}\rho_{kl}\sum_{m=1}^{n}\frac{A_k(i,m)*B_l(m,j)}{x*y*\rho_{kl}}=\sum_{k=1}^{s}\sum_{l=1}^{t}\sum_{m=1}^{n}\frac{A_k(i,m)*B_l(m,j)}{x*y}=\frac{1}{x*y}C(i,j)$*. In summary,* $E(\tilde{C}(i,j))=\sum_{h=1}^{x*y}E(X_h)=\sum_{h=1}^{x*y}\frac{1}{x*y}C(i,j)=C(i,j)$.

THEOREM 2. *Given the same definitions in Theorem 1, then the variance* $Var(\tilde{C}(i,j))=\sum_{k=1}^{s}\sum_{l=1}^{t}\frac{1}{x*y*\rho_{kl}}(\sum_{m=1}^{n}A_k(i,m)*B_l(m,j))^2-\frac{1}{x*y}C(i,j)^2$ *for* $\forall i,j\in\{1,\cdots,n\}$.

Proof. Since $\tilde{C}(i,j)$ *is the sum of* $x*y$ *independent random variables, i.e.,* $\tilde{C}(i,j)=\sum_{h=1}^{x*y}X_h$*, then* $Var(\tilde{C}(i,j))=Var(\sum_{h=1}^{x*y}X_h)=\sum_{h=1}^{x*y}Var(X_h)$*. Also,* $Var(X_h)=E(X_h^2)-E^2(X_h)=\sum_{k=1}^{s}\sum_{l=1}^{t}\rho_{kl}(\sum_{m=1}^{n}\frac{A_k(i,m)*B_l(m,j)}{x*y*\rho_{kl}})^2-(\frac{1}{x*y}C(i,j))^2=\sum_{k=1}^{s}\sum_{l=1}^{t}\frac{1}{x^2*y^2\rho_{kl}}(\sum_{m=1}^{n}A_k(i,m)*B_l(m,j))^2-\frac{1}{x^2*y^2}C(i,j)^2$*. Thus,* $Var(\tilde{C}(i,j))=\sum_{h=1}^{x*y}(\sum_{k=1}^{s}\sum_{l=1}^{t}\frac{1}{x^2*y^2*\rho_{kl}}(\sum_{m=1}^{n}A_k(i,m)*B_l(m,j))^2-\frac{1}{x^2*y^2}C(i,j)^2)=\sum_{k=1}^{s}\sum_{l=1}^{t}\frac{1}{x*y*\rho_{kl}}(\sum_{m=1}^{n}A_k(i,m)*B_l(m,j))^2-\frac{1}{x*y}C(i,j)^2$.

THEOREM 3. *Given the same definitions in Theorem 1,*

$$\min_{\rho_{kl}}\left\{E(\|\tilde{C}-C\|_F^2):\sum_{k=1}^{s}\sum_{l=1}^{t}\rho_{kl}=1\right\}=\frac{1}{x*y}(\sum_{k=1}^{s}\sum_{l=1}^{t}\|A_kB_l\|_F)^2-\frac{1}{x*y}\|C\|_F^2 \text{ when } \rho_{kl}=\frac{\|A_kB_l\|_F}{\sum_{k=1}^{s}\sum_{l=1}^{t}\|A_kB_l\|_F} \quad (3)$$

Proof. According to Theorem 1, $E(\|\tilde{C}-C\|_F^2)=\sum_{i=1}^{n}\sum_{j=1}^{n}E((\tilde{C}(i,j)-C(i,j))^2)=\sum_{i=1}^{n}\sum_{j=1}^{n}(E(\tilde{C}(i,j)^2)-E^2(\tilde{C}(i,j)))=\sum_{i=1}^{n}\sum_{j=1}^{n}Var(\tilde{C}(i,j))$ *since* $E(\tilde{C}(i,j))=C(i,j)=E(C(i,j))$*. From Theorem 2,* $\sum_{i=1}^{n}\sum_{j=1}^{n}Var(\tilde{C}(i,j))=\sum_{i=1}^{n}\sum_{j=1}^{n}(\sum_{k=1}^{s}\sum_{l=1}^{t}\frac{1}{x*y*\rho_{kl}}(\sum_{m=1}^{n}A_k(i,m)*B_l(m,j))^2-\frac{1}{x*y}C(i,j)^2)=\frac{1}{x*y}\sum_{k=1}^{s}\sum_{l=1}^{t}\frac{1}{\rho_{kl}}\|A_kB_l\|_F^2-\frac{1}{x*y}\|C\|_F^2$.

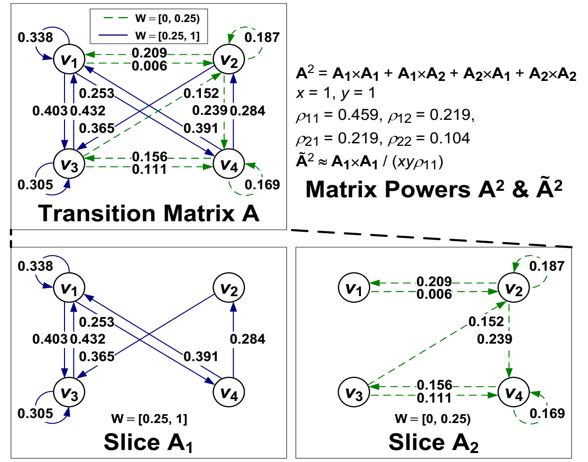


Figure 6: Matrix Power with Slice Pruning

Let $f(\rho_{kl})=\frac{1}{x*y}\sum_{k=1}^{s}\sum_{l=1}^{t}\frac{1}{\rho_{kl}}\|A_kB_l\|_F^2-\frac{1}{x*y}\|C\|_F^2$*, non-zero entries in Hessian* $f''(\rho_{kl})$ *are diagonal entries* $\frac{2}{x*y*\rho_{kl}^3}\|A_kB_l\|_F^2$ *such that* $h^Tf''(\rho_{kl})h\geq 0$ *for* $\forall st\times 1$*-vector* h *in* $\mathbb{R}^{st}$*, i.e.,* $f''(\rho_{kl})$ *is positive-semidefinite.* $f(\rho_{kl})$ *is thus a convex function. In addition, the constraint set* $\{\rho_{kl}|\sum_{k=1}^{s}\sum_{l=1}^{t}\rho_{kl}=1\}$ *is both convex and concave. The minimization problem of a convex function on a convex set is a convex programming problem. Thus, there exists a verifiable sufficient and necessary condition for global optimality. Let* $\rho=[\rho_{11};\cdots;\rho_{1t};\rho_{21};\cdots;\rho_{2t};\cdots;\rho_{s1};\cdots;\rho_{st}]$*, we calculate the global optimal solution by solving the KKT condition.*

$$\nabla_\rho\left(\frac{1}{x*y}\sum_{k=1}^{s}\sum_{l=1}^{t}\frac{1}{\rho_{kl}}\|A_kB_l\|_F^2-\frac{1}{x*y}\|C\|_F^2+\lambda(\sum_{k=1}^{s}\sum_{l=1}^{t}\rho_{kl}-1)\right)=0$$
$$\sum_{k=1}^{s}\sum_{l=1}^{t}\rho_{kl}-1=0 \quad (4)$$

We get $\lambda=\frac{1}{x*y}(\sum_{k=1}^{s}\sum_{l=1}^{t}\|A_kB_l\|_F)^2$ *and the optimal solution* $\rho_{kl}=\frac{\|A_kB_l\|_F}{\sum_{k=1}^{s}\sum_{l=1}^{t}\|A_kB_l\|_F}$*. Therefore, the optimal value of* $E(\|\tilde{C}-C\|_F^2)$ *is equal to* $\frac{1}{x*y}(\sum_{k=1}^{s}\sum_{l=1}^{t}\|A_kB_l\|_F)^2-\frac{1}{x*y}\|C\|_F^2$.

However, A_kB_l represents the multiplication between slices A_k and B_l. Thus, we can not get such ρ_{kl} before doing slice multiplication. Notice that $\|A_kB_l\|_F^2=\sum_{i=1}^{n}\|A_kB_l(i,:)\|_2^2=\sum_{i=1}^{n}\|A_k(i,:)B_l\|_2^2\leq\sum_{i=1}^{n}\|A_k(i,:)\|_2^2\|B_l\|_F^2=\|A_k\|_F^2\|B_l\|_F^2\Rightarrow\|A_kB_l\|_F\leq\|A_k\|_F\|B_l\|_F$ where $A_kB_l(i,:)$ represents the ith row of multiplication between two slices, and $A_k(i,:)$ denotes the ith row of A_k. Thus, we use the *SliceDensity* measure in Eq.(2) to generate a near-optimal $\rho_{kl}=\frac{\|A_k\|_F\|B_l\|_F}{\sum_{i=1}^{s}\sum_{j=1}^{t}\|A_i\|_F\|B_j\|_F}$ in line 4 in Algorithm 2.

Figure 6 presents an example of slice pruning for computing the square of the transition matrix $\mathbf{A}$ of a given graph. We partition $\mathbf{A}$ into two equal-size slices: $\mathbf{A_1}$ with edge weights of $[0.25,1]$ and $\mathbf{A_2}$ with edge weights of $[0,0.25)$. The exact matrix power $\mathbf{A}^2$ is equal to $\sum_{k=1}^{2}\sum_{l=1}^{2}A_k\times A_l$. If we

Algorithm 3 **StripMultiply**(A, k, x, B, l, y, z)

1: Initialize $E[1 \cdots \lfloor n/q \rfloor][1 \cdots \lfloor n/q \rfloor]$;
2: **for** $m = 1, \cdots, n$
3: $\quad \rho_m = \frac{CutDensity(A_{kxm}) * CutDensity(B_{lym})}{\sum_{i=1}^{n} CutDensity(A_{kxi}) * CutDensity(B_{lyi})}$;
4: **for** $h = 1, \cdots, z$ independently
5: $\quad$ Pick $a \in \{1, \cdots, n\}$ with Prob$(a = m) = \rho_m$, $m = 1, \cdots, n$;
6: $\quad \Phi = \Phi \cup \{a\}$;
7: $\tilde{M}$ = **PartitionLoad**$(A, k, x, out, strip, \Phi)$;
8: $\tilde{N}$ = **PartitionLoad**$(B, l, y, in, strip, \Phi)$;
9: R = **ParallelJoin**$(\tilde{M}, \tilde{N}, \tilde{M}.D = \tilde{N}.S)$
10: **for each record** in R
11: $\quad E[a.source][b.destination].AddWeight(a.weight * b.weight/ (z * \rho_m))$ where m is the index of $a.destination$ and $b.source$;
12: **return** E;

want to use only one slice multiplication to approximate $\mathbf{A}^2$, then it is very probable that $\mathbf{A_1} \times \mathbf{A_1}$ is picked to approximate $\mathbf{A}^2$ with rescaling factor $\frac{1}{x*y*\rho_{11}}$ since ρ_{11} is maximal.

3.2 Cut Pruning (Vertex-based Pruning)

Alternative to the slice pruning at the partition (subgraph) level, the cut pruning at the vertex level is to prune some insignificant vertices with their associated edges by using cut density as a statistical measure to evaluate the significance of a vertex cut. Similar to matrix multiplication, a fast Monte-Carlo approach is used to further improve the performance of strip multiplications with bounded error.

Definition 2. *[CutDensity] Let* $K{=}(S, D_K, E_K, W_J)$ *be a strip of an in-edge slice* $J{=}(S, D, E_J, W_J)$ *defined in Eq.(2) where* $D_K{\subseteq}D$ *and* $E_K{=}\{(u,v)|u{\in}S, v{\in}D_K, (u,v).weight \in W_J, (u,v){\in}E_J\}$*, and* $L(u){=}(u, D_K, E_L, W_J)$ *be an out-edge cut of* K *where* $u{\in}S$ *and* $E_L{=}\{(u,v)|v{\in}D_K, (u,v){\in}E_K, (u,v).weight{\in}W_J\}$*, the density of* $L(u)$ *is defined as follow.*

$$CutDensity(L(u)) = |L(u)| = \sqrt{\sum_{v \in D_K} \sum_{(u,v) \in E_L} w(u,v)^2} \qquad (5)$$

where $L(u)$ *denotes the out-edge cut itself and its (row) vector representation,* $|L(u)|$ *is the magnitude of vector* $L(u)$*, and* $w(u,v)$ *denotes the weight of an edge* (u,v)*. If an out-edge cut has many edges with large weights, then it will have a large density. The density definition of an in-edge cut, corresponding to a column vector, is similar to Eq.(5).*

Given an out-edge strip M (i.e., A_{kx}: the xth strip in slice A_k) with n in-edge cuts $M_1, \cdots, M_n$ (i.e., cuts $A_{kx1}, \cdots, A_{kxn}$), and an in-edge strip N (i.e., strip B_{ly}) with n out-edge cuts $N_1, \cdots, N_n$ (i.e., cuts $B_{ly1}, \cdots, B_{lyn}$), we decompose a strip multiplication $O{=}M{\times}N$ into n cut multiplications, i.e., $O{=}\sum_{m=1}^{n} M_m{\times}N_m$. Similarly, the strip multiplication with cut pruning reduces the exact n cut multiplications to the approximate z ($<< n$) cut multiplications, while maintaining the utility with bounded error. The selection probability ρ_m of each pair of cuts (M_m and N_m) is first calculated in terms of their *CutDensity*. The algorithm independently performs z trials to choose z "significant" cut pairs from the original n cut pairs. Φ maintains the z index entries for chosen cuts of M and N in z trials. We then invoke the *PartitionLoad* routine to load and filter the out-edge strip M and the in-edge strip N with the chosen cuts. The *ParallelJoin* routine is executed at the cut level to join in-edges (a) and out-edges (b) of z "significant" vertices. The final edge set E is produced by summarizing the pairwise-cut multiplications with rescaling factor $\frac{1}{z*\rho_m}$

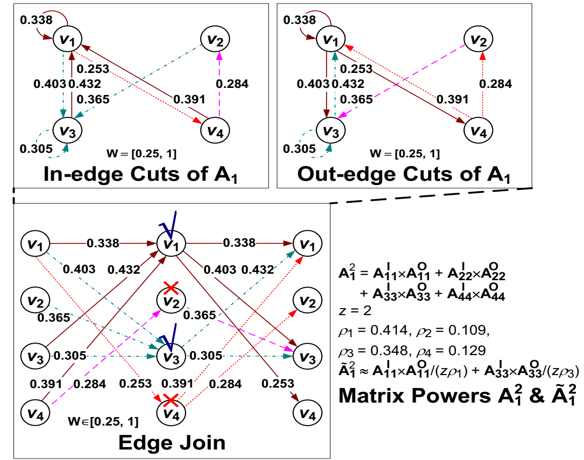


Figure 7: Matrix Power with Cut Pruning

to compensate for the cuts that are not picked. Thus, the multiplication is approximated as $\tilde{O} = \sum_{h=1}^{z} \frac{M_{\Phi(h)} \times N_{\Phi(h)}}{z * \rho_{\Phi(h)}}$.

Similarly, the following theoretical results demonstrate that our cut pruning strategy can achieve a good approximation with bounded error.

Theorem 4. *Given an actual multiplication* $O{=}\sum_{m=1}^{n} M_m{\times}N_m$*, a multiplication approximation* $\tilde{O}{=}\sum_{h=1}^{z} \frac{M_{\Phi(h)} \times N_{\Phi(h)}}{z*\rho_{\Phi(h)}}$*, then the expectation* $E(\tilde{O}(i,j)){=}O(i,j)$ *for* $\forall i,j{\in}\{1,\cdots,n\}$.

Theorem 5. *Given the same definitions in Theorem 4, then the variance* $Var(\tilde{O}(i,j)){=}\sum_{m=1}^{n} \frac{M(i,m)^2 * N(m,j)^2}{z*\rho_m} {-} \frac{1}{z} O(i,j)^2$ *for* $\forall i,j{\in}\{1,\cdots,n\}$.

Theorem 6. *Given the same definitions in Theorem 4,*

$$\min_{\rho_m} \left\{ E(\|\tilde{O} - O\|_F^2) : \sum_{m=1}^{n} \rho_m = 1 \right\} = \frac{1}{z}\left(\sum_{m=1}^{n} |M_m||N_m|\right)^2 - \frac{1}{z}\|O\|_F^2$$
$$when\ \rho_m = \frac{|M_m||N_m|}{\sum_{m=1}^{n} |M_m||N_m|} \qquad (6)$$

The proof of Theorems 4-6 is omitted due to space limit. The proof methods are similar to that used in Theorems 1-3.

Figure 7 shows an example of cut pruning for computing the square of slice $\mathbf{A_1}$ in Figure 6. Slice $\mathbf{A_1}$ is organized as 4 in-edge cuts and 4 out-edge cuts respectively. Ochre edges, purple edges, green edges and red edges represent in-edge cuts and out-edge cuts of vertices v_1, v_2, v_3 and v_4, respectively. The exact $\mathbf{A_1}^2$ is equal to the sum of 4 cut multiplications: $\mathbf{A^I_{11}} {\times} \mathbf{A^O_{11}}$ (ochre edge join), $\mathbf{A^I_{12}} {\times} \mathbf{A^O_{12}}$ (purple edge join), $\mathbf{A^I_{13}} {\times} \mathbf{A^O_{13}}$ (green edge join), and $\mathbf{A^I_{14}} {\times} \mathbf{A^O_{14}}$ (red edge join) where $\mathbf{A^I_{1m}}$ and $\mathbf{A^O_{1m}}$ represent the mth in-edge cut and the mth out-edge cut for vertex v_m respectively. If we want to use only two cut multiplications to approximate $\mathbf{A}_1^2$, then it is highly probable that $\mathbf{A^I_{11}} {\times} \mathbf{A^O_{11}}$ and $\mathbf{A^I_{13}} {\times} \mathbf{A^O_{13}}$ are picked with rescaling factors $\frac{1}{z*\rho_1}$ and $\frac{1}{z*\rho_3}$ since ρ_1 and ρ_3 are maximal. The cut pruning stops graph propagation through low degree vertices and edges with small weights.

4. EXPERIMENTAL STUDY

We use several typical iterative graph applications to evaluate the performance of graph processing systems on a

Graph	#Vertices	#Edges	AvgDeg	MaxIn	MaxOut	DegDist
RMAT	4.3B	17.2B	4	1.8M	737.4K	power-law
ErdosRenyi	1.1B	13.0B	12.1	259.7K	434.8K	Poisson
Random	330M	2.1B	6.5	100K	85.6K	heavy-tailed
Kronecker	88M	1.5B	17.5	365.2K	19.3K	multinomial

Table 2: Synthetic Simple Graph Datasets

Application	Graph Type	Core Computation
PageRank [7]	single graph	matrix-vector
SpMV [5]	single graph	matrix-vector
Connected Components [36]	single graph	graph traversal
Matrix Power	two graphs	matrix-matrix
Diffusion Kernel [18]	two graphs	matrix-matrix
AEClass [34]	multigraph	matrix-vector

Table 3: Graph Applications

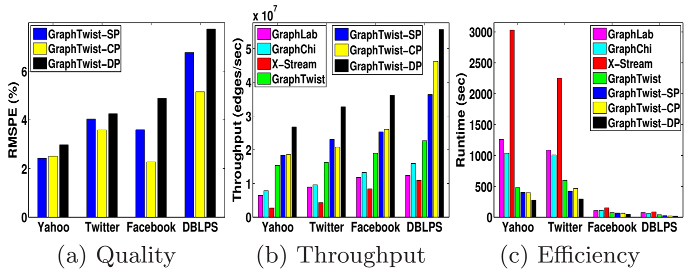
set of real-world graphs in Table 1 and synthetic graphs in Table 2. DBLPS is a single coauthor graph. DBLPM is a coauthor multigraph, where each pair of authors have at most 24 parallel coauthor links, each corresponding to one of 24 research fields, as mentioned in Section 2.3. Similarly, we build a friendship multigraph of Last.fm, where each friendship edge is classified into a subset of 21 music genres in terms of the same artists shared by two users. Based on the Recursive Matrix (R-MAT) model [8], we utilize the GTgraph suite [4] to generate a scale-free small-world graph with power-law degree distribution. We also use the GTgraph suite [4] based on the Erdos-Renyi random graph model [11] to produce a large-scale random network with Poisson degree distribution. The GenGraph generator [28] is used to construct a large random simple connected graph with heavy-tailed degree distribution. In addition, we employ the SNAP Krongen tool [21] to generate a stochastic Kronecker graph through the Kronecker product. All experiments were performed on a 4-core PC with Intel Core i5-750 CPU at 2.66 GHz, 16 GB memory, and a 1 TB hard drive, running Linux 64-bit. We compare GraphTwist with three existing graph parallel models: **GraphLab** [22], **GraphChi** [20] and **X-Stream** [25]. To evaluate the effectiveness of pruning strategies, we evaluate the following versions of GraphTwist: (1)**GraphTwist** with only access tier abstractions; (2) **GraphTwist-SP** which improves GraphTwist with slice pruning; (3) **GraphTwist-CP** which enhances GraphTwist with cut pruning; and (4) **GraphTwist-DP** which uses both pruning optimizations.

4.1 Evaluation Measures

We evaluate the performance of graph processing systems by measuring the running time and the *Throughput* (the number of edges processed per second). We adopt the root-mean-square-percentage-error (RMSPE) between the actual result and the approximate result to evaluate the quality of three versions of GraphTwist with pruning strategies.

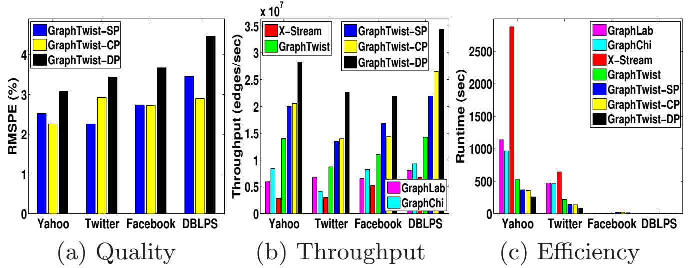
$$RMSPE(X, \hat{X}) = \sqrt{\frac{\sum_{i=1}^{n}((x_i - \hat{x}_i)/x_i)^2}{n}}$$
$$RMSPE(X, \hat{X}) = \sqrt{\frac{\sum_{i=1}^{n}\sum_{j=1}^{n}((x_{ij} - \hat{x}_{ij})/x_{ij})^2}{n^2}} \quad (7)$$

where x_i is a component in the resulted vector X by the exact computation (matrix-vector computation) and $\hat{x}_i$ is an entry in the approximate vector $\hat{X}$ by the computation with pruning strategies. Similarly, x_{ij} is an element in the actual result matrix X by matrix-matrix computation and



(a) Quality (b) Throughput (c) Efficiency

Figure 8: PageRank on Four Simple Graphs



(a) Quality (b) Throughput (c) Efficiency

Figure 9: SpMV on Four Simple Graphs

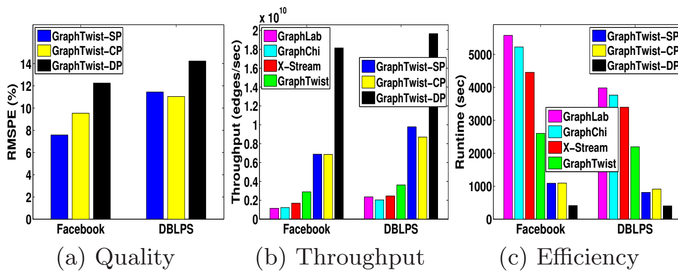
$\hat{x}_{ij}$ is an entry in the approximate result matrix $\hat{X}$. A lower RMSPE number indicates a better performance.

4.2 Execution Efficiency on Single Graph

Figures 8 and 9 present the quality and performance comparison of iterative algorithms on four single graphs by different graph processing systems with different scales: Yahoo, Twitter, Facebook and DBLPS with $\#iterations$=1, 5, 40, 30 respectively. Since GraphLab, GraphChi, X-Stream and GraphTwist are the exact graph computations without pruning optimizations, they always have a RMSPE value of zero. Thus, we do not plot the RMSPE bars for four exact graph computations. Figure 8 (a) shows the RMSPE values by GraphTwist with different pruning strategies. GraphTwist-DP achieves the highest RMSPE values since it adopts a dual pruning scheme to achieve the highest efficiency. GraphTwist-CP gains a much lower RMSPE than GraphTwist-SP. The RMSPE values by three approximate systems are smaller than 7.8%, even with $\#iterations$=40. This demonstrates that applying pruning techniques to iterative graph applications can achieve a good approximation.

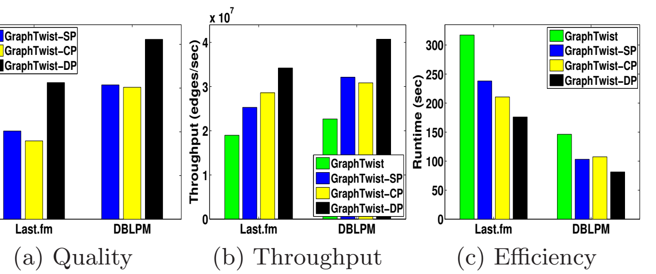
Figure 8 (b) exhibits the throughput comparison on four datasets. The throughput values by the exact GraphTwist are larger than 1.53×10^7 and consistently higher than GraphLab, GraphChi and X-Stream. All versions of GraphTwist with pruning turned on significantly outperform the exact GraphTwist, with the throughput by GraphTwist-DP as the highest ($>2.67 \times 10^7$). GraphTwist-SP and GraphTwist-CP achieve slightly lower throughput than GraphTwist-DP.

Figure 8 (c) compares the running time by different graph parallel models, from loading graph from disk to writing results back to disk. The runtime comparison is consistent with the throughput evaluation in Figure 8 (b). We make two interesting observations. First, the exact GraphTwist outperforms GraphLab, GraphChi and X-Stream in all experiments. Second, GraphTwist with different pruning strategies significantly outperform the exact GraphTwist in



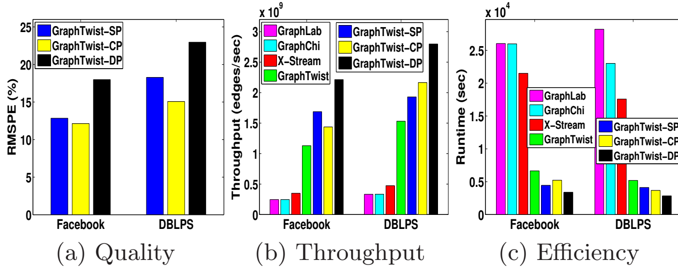
(a) Quality (b) Throughput (c) Efficiency

Figure 10: Matrix Power on Two Simple Graphs



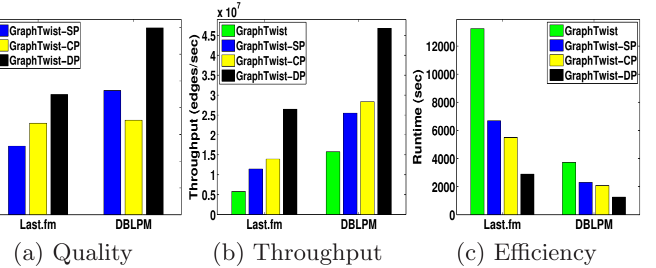
(a) Quality (b) Throughput (c) Efficiency

Figure 12: PageRank on Multigraph



(a) Quality (b) Throughput (c) Efficiency

Figure 11: Diffusion Kernel on Two Simple Graphs



(a) Quality (b) Throughput (c) Efficiency

Figure 13: AEClass on Multigraph

terms of throughput and runtime while maintaining very good quality in terms of RMSPE.

Similar trends are observed for the performance comparison of SpMV in Figure 9. Compared to GraphLab, GraphChi and X-Stream, the exact GraphTwist consistently performs better in all throughput and efficiency tests. All versions of GraphTwist with pruning significantly outperform the exact GraphTwist thanks to the pruning optimizations while maintaining a good approximation.

4.3 Execution Efficiency on Multiple Graphs

Figures 10 and 11 show the performance comparison of iterative applications on multiple graphs with different graph parallel models. Since GraphLab, GraphChi and X-Stream can not directly address matrix-matrix multiplications among multiple graphs, we thus modify the corresponding implementations to run the above graph applications. As the complexity of matrix-matrix multiplication ($O(n^3)$) is much larger than the complexity of matrix-vector multiplication ($O(n^2)$), we only compare the performance by different graph processing systems on two smaller datasets: Facebook and DBLPS. We observe the very similar trends as those shown in Figures 8 and 9. All versions of GraphTwist significantly prevail over GraphLab, GraphChi and X-Stream in all efficiency tests, and GraphTwist-DP with double pruning strategies obtains the highest throughput.

4.4 Execution Efficiency on Multigraph

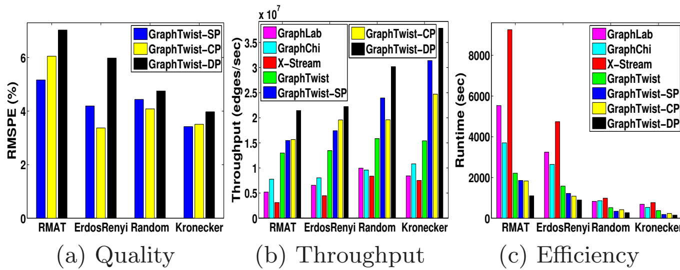
Since existing representative graph processing systems can not address iterative applications on multigraphs, we only perform the efficiency comparison of PageRank on multigraphs by different GraphTwist versions, as shown in Figure 12. GraphTwist partitions the 3D cube of a multigraph into p slices along dimension W. Each slice consists of parallel edges with a unique semantics, say the DBLP coauthored papers in the area of DB and the Last.fm user friendships with respect to pop music genre. By hashing the parallel edges with the same semantics into the same partition, each slice corresponds to one partition, and represents a subgraph with only those edges that have the corresponding semantics included in the hash bucket for that partition. The PageRank algorithm is executed on each slice with a unique semantics in parallel to compute the ranking vector of authors (or users) in each research field (or music genre). Figure 13 presents the performance comparison of a multigraph algorithm (AEClass) with the GraphTwist implementation. AEClass [34] transforms the problem of multi-label classification of heterogeneous networks into the task of multi-label classification of coauthor (or friendship) multigraph based on activity-based edge classification. We observe very similar trends as those shown in Figures 8-11. All three GraphTwist versions with pruning achieve relatively lower RMSPE ($<$14.0%). GraphTwist-DP obtains the highest throughput ($>2.6\times10^7$), while the throughput by the exact GraphTwist is the lowest ($>5.7\times10^6$). The throughput values by GraphTwist-SP and GraphTwist-CP stand in between ($>1.1\times10^7$).

4.5 Execution Efficiency on Synthetic Graphs

Figure 14 shows the performance comparison of GraphTwist with the other three systems by PageRank on four synthetic graphs with $\#iterations$=2, 2, 5, 5 respectively. We have observed similar trends to the performance on real-world graphs. GraphTwist consistently outperforms GraphLab, GraphChi and X-Stream in both throughput and efficiency tests. GraphTwist with all versions of pruning significantly outperform the exact GraphTwist thanks to the pruning optimizations while maintaining a good approximation.

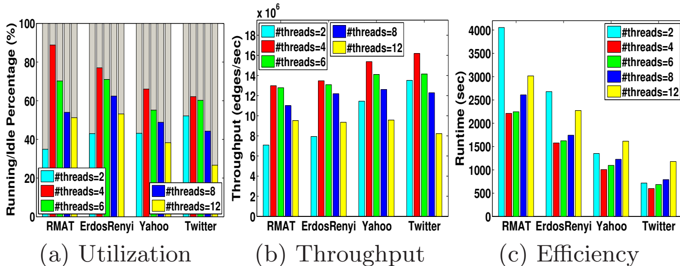
4.6 Impact of #Threads

Figure 15 presents the performance comparison of GraphTwist by running PageRank on two synthetic simple graphs and two real simple graphs with $\#iterations$=2, 2, 3, 5 respectively. Figure 15 (a) shows the average utilization com-



(a) Quality (b) Throughput (c) Efficiency

Figure 14: PageRank on Four Synthetic Simple Graphs



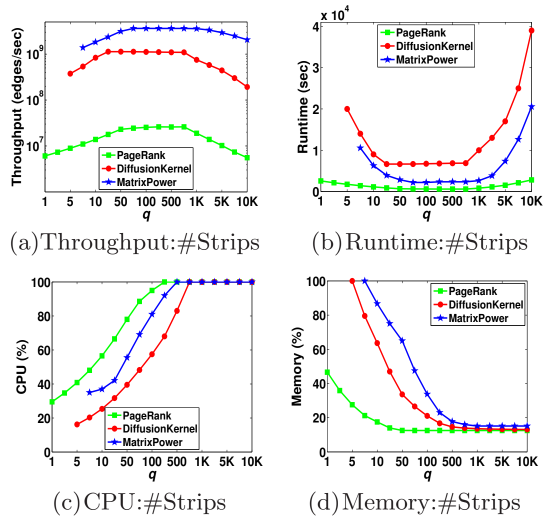
(a) Utilization (b) Throughput (c) Efficiency

Figure 15: PageRank by GraphTwist wrt Varying Threads

parison (computation/non-computation runtime percentage) of each thread by varying *#threads* from 2, 4, 6, 8, to 12. Recall Section 2.8, two levels of threads are used in GraphTwist: partition-level threads to process multiple partitions (slices, dices or strips) in parallel. Within each partition thread, we also execute multiple threads at vertex level. The total number of threads is arranged by $h{*}l$ (h is *#threads* at partition level and l is *#threads* at vertex level). For example, for the setup of 8 threads, we use 2 threads at partition level and 4 threads at vertex level. We compare the total runtime to execute a task of iterative graph computation, including thread running time and thread idle runtime. The thread idle runtime is measured when performing non-computation operations, including thread waiting, disk I/O, context switching and CPU scheduling. Figure 15 (a) shows that the utilization rate on all four graphs with 4, 6 or 8 threads are better compared to 2 or 12 threads. When *#threads*=2, the threads are busy at <57% of time and when *#threads*=12, the threads are idle at >46% of time. In both cases, the threads do not efficiently utilize CPU resource. Figure 15 (c) (or Figure 15 (b)) measure the performance impact of parallel threads by PageRank on four simple graphs. We have observed that the runtime is very long when the number of parallel threads is relatively small (*#threads*=2) or very large (*#threads*=12) and it is almost a stable horizontal line when *#threads*=4, 6, 8). Also GraphTwist usually achieves the best performance by spawning between *#cores* and 2∗*#cores* threads because less *#threads* (<*#cores*) often lead to underutilization of available CPU resource in graph parallel system. On the other hand, more threads (>2∗*#cores*) may introduce additional non-computation overhead, such as context switching and CPU scheduling, and thus hurt system performance.

4.7 Decision of #Partitions

Figure 16 measures the performance impact of different numbers of strips on GraphTwist with PageRank over Twitter, Diffusion Kernel on Facebook, and Matrix Power on DBLPS. The x-axis shows different settings of the number of strips. We vary *#Strips* from 1 to 10,000 and fix *#Slices* and *#Dices* as 5 in each figure. It is observed that the runtime curve (or the throughput curve) for each application in each figure follows a similar "U" curve (inverted "U" curve) with respect to the size of strip, i.e., the runtime is very long when the unit size is relatively small or very large and it is almost a stable horizontal line when the unit size stands in between two borderlines. This is because bigger strips often lead to substantial work imbalance in graph applications. On the other hand, smaller strips may result in frequent external storage access and lots of page replacements between units lying in different pages. Figure 16 (c) measures the CPU utilization by GraphTwist for three real applications. The CPU utilization rate of each application increases quickly when *#Strips* is increasing. This is because for the same graph, the larger number of strips gives the smaller size per strip and the smaller strips in big graphs often lead to better workload balancing for parallel computations. Figure 16 (d) shows the memory utilization comparison. The memory curves are totally contrary to the corresponding CPU curves: the smaller the number of strips, the larger size each strip will have, thus the larger the memory usage. The performance impact of *#Slices* or *#Dices* on GraphTwist have similar trends to Figure 16.



(a)Throughput:#Strips (b)Runtime:#Strips

(c)CPU:#Strips (d)Memory:#Strips

Figure 16: Impact of #Strips

5. RELATED WORK

Graph parallel abstraction has been a heated topic in recent years. Research activities can be classified into two broad categories below [3, 13–15, 20, 22–25, 27, 30].

Given that GraphTwist is a single PC based solution, it is more relevant to the previous works in this category [15, 20, 22, 25, 30]. GraphLab [22] presented a new sequential shared memory abstraction where each vertex can read and write data on adjacent vertices and edges. It supports the representation of structured data dependencies and flexible scheduling for iterative computation. GraphChi [20] partitions a graph into multiple shards by storing each vertex and its all in-edges in one shard. It introduces a novel parallel sliding window based method to facilitate fast access

to the out-edges of a vertex stored in other shards. TurboGraph [15] improves GraphChi by storing a large graph in the slotted page list with each page containing a number of adjacency lists. It utilizes the parallelism of both multi-core CPU and Flash SSD I/O to improve the access efficiency. X-Stream [25] is an edge-centric approach to the scatter-gather model on a single shared-memory machine and it uses CPU cache and multi-threading to improve the access efficiency.

Distributed graph systems have attracted active research in recent years [3,13,14,23,24,27]. Pregel [24] is a bulk synchronous message passing abstraction where vertices can receive messages sent in the previous iteration, send messages to other vertices and modify its own state and that of its outgoing edges or mutate graph topology. PowerGraph [13] extends GraphLab [22] and distributed GraphLab [23] by using the Gather-Apply-Scatter model of computation to address the natural graphs with highly skewed power-law degree distributions. GraphX [14] enables iterative graph computation, written in Scala like API in terms of GraphX RDG, to run on the SPARK cluster platform.

6. CONCLUSION

We present a scalable, efficient, provably correct two-tier graph parallel processing system, GraphTwist. At storage and access tier, GraphTwist employs three customizable parallel abstractions: slice partitioning, strip partitioning and dice partitioning, to maximize parallel computation efficiency. At computation tier, GraphTwist presents slice pruning and cut pruning strategies. Our pruning methods are utility-aware and can significantly speed up the computational performance while preserving the computational utility defined by the graph applications.

7. ACKNOWLEDGMENTS

This work was performed under the partial support by the NSF CISE under Grants IIS-0905493, CNS-1115375, IIP-1230740 and a grant from Intel ISTC on Cloud Computing.

8. REFERENCES

[1] http://www.informatik.uni-trier.de/∼ley/db/.
[2] http://www.last.fm/api/.
[3] Giraph. http://giraph.apache.org/.
[4] D. A. Bader and K. Madduri. Gtgraph: A synthetic graph generator suite, 2006.
[5] M. Bender, G. Brodal, R. Fagerberg, R. Jacob, and E. Vicari. Optimal sparse matrix dense vector multiplication in the i/o-model. *Theory of Computing Systems*, 47(4):934–962, 2010.
[6] O. Bretscher. *Linear Algebra With Applications, 3rd Edition*. Prentice Hall, 1995.
[7] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *WWW*, 107–117, 1998.
[8] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, 442–446, 2004.
[9] T. Chakraborty, S. Sikdar, V. Tammana, N. Ganguly, and A. Mukherjee. Computer science fields as ground-truth communities: Their impact, rise and fall. In *ASONAM*, pages 426–433, 2013.
[10] J. Cohen, P. Cohen, S. G. West, and L. S. Aiken. *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences, 3rd Edition*. Routledge, 2002.
[11] P. Erdos and A. Renyi. On random graphs i. *Publicationes Mathematicae*, 6:290–297, 1959.
[12] M. Gjoka, M. Kurant, C. T. Butts, and A. Markopoulou. Walking in facebook: A case study of unbiased sampling of osns. In *INFOCOM*, pages 2498–2506, 2010.
[13] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 17–30, 2012.
[14] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, 599–613, 2014.
[15] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu. Turbograph: A fast parallel graph engine handling billion-scale graphs in a single pc. In *KDD*, pages 77–85, 2013.
[16] F. S. Hillier and G. J. Lieberman. *Introduction to Operations Research (IBM)*. Mcgraw-Hill College, 1995.
[17] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. In *WWW*, pages 640–651, 2003.
[18] R. I. Kondor and J. D. Lafferty. Diffusion kernels on graphs and other discrete input spaces. In *ICML*, 315–322, 2002.
[19] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *WWW*, 591–600, 2010.
[20] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *OSDI*, pages 31–46, 2012.
[21] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani. Kronecker graphs: An approach to modeling networks. *JMLR*, 11:985–1042, 2010.
[22] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. In *UAI*, pages 340-349, 2010.
[23] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *PVLDB*, 5(8):716–727, 2012.
[24] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, 135–146, 2010.
[25] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *SOSP*, pages 472-488, 2013.
[26] V. Satuluri and S. Parthasarathy. Scalable graph clustering using stochastic flows: Applications to community discovery. In *KDD*, pages 737-746, 2009.
[27] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *SIGMOD*, 505–516, 2013.
[28] F. Viger and M. Latapy. Efficient and simple generation of random simple connected graphs with prescribed degree sequence. In *COCOON*, pages 440-449, 2005.
[29] Y. Webscope. Yahoo! altavista web page hyperlink connectivity graph, circa 2002. http://webscope.sandbox.yahoo.com/.
[30] W. Xie, G. Wang, D. Bindel, A. Demers, and J. Gehrke. Fast iterative graph computation with block updates. *PVLDB*, 6(14):2014–2025, 2013.
[31] Y. Zhou, H. Cheng, and J. X. Yu. Graph clustering based on structural/attribute similarities. *PVLDB*, 2(1):718–729, 2009.
[32] Y. Zhou, H. Cheng, and J. X. Yu. Clustering large attributed graphs: An efficient incremental approach. In *ICDM*, pages 689–698, 2010.
[33] Y. Zhou and L. Liu. Social influence based clustering of heterogeneous information networks. In *KDD*, pages 338–346, 2013.
[34] Y. Zhou and L. Liu. Activity-edge centric multi-label classification for mining heterogeneous information networks. In *KDD*, pages 1276–1285, 2014.
[35] Y. Zhou, L. Liu, K. Lee, C. Pu, and Q. Zhang. Fast iterative graph computation with resource aware graph parallel abstractions. In *HPDC*, 2015.
[36] X. Zhu and Z. Ghahramani. Learning from labeled and unlabeled data with label propagation. In *CMU CALD Tech Report*, 2002.