Fast Iterative Graph Computation with Resource Aware Graph Parallel Abstractions

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

ABSTRACT

Iterative computation on large graphs has challenged system research from two aspects: (1) how to conduct high performance parallel processing for both in-memory and out-of-core graphs; and (2) how to handle large graphs that exceed the resource boundary of traditional systems by resource aware graph partitioning such that it is feasible to run large-scale graph analysis on a single PC. This paper presents GraphLego, a resource adaptive graph processing system with multi-level programmable graph parallel abstractions. GraphLego is novel in three aspects: (1) we argue that vertex-centric or edge-centric graph partitioning are ineffective for parallel processing of large graphs and we introduce three alternative graph parallel abstractions to enable a large graph to be partitioned at the granularity of subgraphs by slice, strip and dice based partitioning; (2) we use dice-based data placement algorithm to store a large graph on disk by minimizing non-sequential disk access and enabling more structured in-memory access; and (3) we dynamically determine the right level of graph parallel abstraction to maximize sequential access and minimize random access. GraphLego can run efficiently on different computers with diverse resource capacities and respond to different memory requirements by real-world graphs of different complexity. Extensive experiments show the competitiveness of GraphLego against existing representative graph processing systems, such as GraphChi, GraphLab and X-Stream.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Reliability, availability, and serviceability; D.1.3 [**Concurrent Programming**]: Parallel Programming

Keywords

Graph Processing System; Large-scale Graph; Parallel Computing; 3D Cube Representation; Multigraph Processing

1. INTRODUCTION

Scaling iterative computation on large graphs with billions of vertices and billions of edges is widely recognized as a challenging systems research problem, which has received heated attention recently [1–20]. We can classify existing research activities into two broad categories: (1) Distributed solutions and (2) Single PC based solutions. Most of existing research efforts are dedicated to the distributed graph partitioning strategies [2, 17–19] to distribute large graphs across a cluster of computer nodes. Several recent efforts [4, 11, 14, 16, 20] have successfully demonstrated huge opportunities for optimizing graph processing on a single PC through efficient storage organization and in-memory computation. However, most existing approaches rely on vertex-centric graph parallel computation model.

Many existing algorithms fail to work effectively under the vertex-centric computation model for several scenarios: (1) when the algorithms require to load the whole graph into the main memory but the graph and its intermediate results of computation together are too big to fit into the available memory; (2) when high degree vertices and their edges combined with the necessary intermediate results are too big to fit into the working memory; (3) when the time of computing on a vertex and its edges is much faster than the time to access to the vertex state and its edge data in memory or on disk; and (4) when the computation workloads on different vertices are significantly imbalanced due to the highly skewed vertex degree distribution.

To address the above issues, we propose to exploit computational parallelism by introducing three alternative graph parallel abstractions that offer larger level of granularity than vertex-centric model, with three complimentary objectives: (1) large graphs should be stored physically on disk using the right level of partition granularity to maximize sequential I/Os; (2) large graphs should be partitioned into graph parallel units such that each partition plus the necessary intermediate results will fit into the working memory and parallel computation on partitions generate well balanced workloads; and (3) different graph parallel abstractions are critical for scaling parallel processing of large graphs to computers with different capacities.

In this paper we present GraphLego, a resource aware graph processing system with multi-level programmable parallel abstractions. GraphLego by design has three novel features. First, to introduce graph parallel abstractions at different levels of granularity, we model a large graph as a 3D cube with source vertex, destination vertex and edge weight as the dimensions. This data structure enables us to introduce multi-level hierarchical graph parallel abstraction by slice, strip and dice based graph partitioning. Second, we dynamically determine the right level of graph parallel abstraction based on the available system resource and the characteristics of graph datasets for each of the iterative graph computation algorithms. By employing flexible and tunable graph parallel abstractions, GraphLego can run iterative graph computations efficiently on any single PC with different CPU and memory capacities. We show that by choosing the right level of parallel abstraction, we can maximize sequential access and minimize random access. Third but not the least, GraphLego uses dice-based data placement algorithm to store a large graph on disk by minimizing non-sequential disk access and enabling more structured in-memory access. By supporting multi-level programmable graph parallel abstractions and dynamic customization, GraphLego enables data scientists to tailor their graph computations in response to different real-world graphs of varying sizes/complexity and different computing platforms with diverse computing resources.

2. RELATED WORK

We classify existing research activities on graph processing system into two broad categories below [1–20].

Single PC based systems [4, 11, 14–16, 20] are gaining attention in recent years. GraphLab [4] 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 [11] partitions a graph into multiple shards by storing each vertex and its 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 [14] presented a multi-thread graph engine by using a compact storage of slotted page list and exploiting the full parallelism of multi-core CPU and Flash SSD I/O. X-Stream [16] is an edge-centric approach to the scatter-gather programming model on a single shared-memory machine. It uses streaming partitions to utilize the sequential streaming bandwidth of the storage medium for graph processing.

Distributed graph systems [1, 2, 5, 8, 10, 13, 17–19] have attracted active research in recent years, with Pregel [2], PowerGraph [10]/Distributed GraphLab [8], and GraphX [19] as the most popular systems. Pregel [2] 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 [10] extends GraphLab [4] and distributed GraphLab [8] by using the Gather-Apply-Scatter model of computation to address the natural graphs with highly skewed power-law degree distributions. GraphX [19] enables iterative graph computation, written in Scala like API in terms of GraphX RDG, to run on the SPARK cluster platform, making the programming of iterative graph algorithms on Spark easier than PowerGraph and Pregel.

Iterative graph applications has been extensively studied in the areas of machine learning, data mining and information retrieval [21, 24–39]. Typical examples of real-world iterative graph applications include ranking, similarity search, graph classification, graph clustering, and collaborative filtering. Popular iterative graph applications can be categorized into three classes in terms of the core computation used in the respective algorithms: (1) matrix-vector computation, such as PageRank [21], EigenTrust [28] and Random Walk with Restart [29]; (2) matrix-matrix computation, including Heat Diffusion Kernel [24, 30], Label Propagation [31], wvRN [32], Markov Clustering [33] and SA-Cluster [34]; and (3) matrix factorization, such as NMF [36], SVD++ [37], Social Regularization [38]. They often need to repeatedly self-interact on a single graph or iteratively interact among multiple graphs to discover both direct and indirect relationships between vertices.

To our best knowledge, GraphLego is the first one to support multi-level programmable parallel graph abstractions (slice, strip, dice) and to provide resource adaptive selection of the right level of graph parallel granularity for partitioning, storing and accessing large graphs.

3. GRAPHLEGO APPROACH

Real graphs often have skewed vertex degree distribution and 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 processing imbalance in parallel computation. In addition, different types of iterative graph applications combined with different sizes of graphs often have different resource demands on CPU, memory and disk I/O. GraphLego is designed to address these issues by introducing resource-adaptive and multi-level programmable graph parallel abstractions.

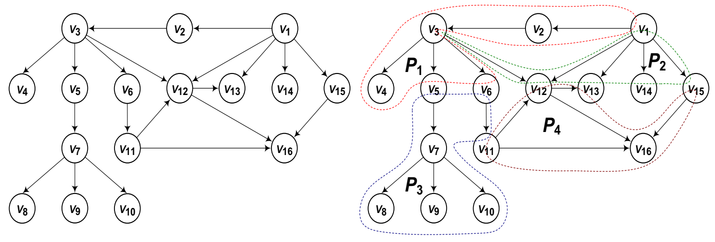
3.1 Graph Processing with 3D Cube

3D Cube. GraphLego represents a given graph G as a 3D cube I with source vertices, destination vertices and edge weights as the three dimensions. Formally, a directed graph is defined as $G=(V, E, W)$ where V is a set of n vertices, 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. Two vertices may be connected by multiple parallel edges. For an edge $e=(u, v) \in E$, we refer to e as the in-edge of v and the out-edge of u and we refer to u and v as the source vertex and the destination vertex of e respectively. In GraphLego, we model a graph G with 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. Given a vertex $u \in S$ and a vertex $v \in D$, if $(u, v) \in E$ then $(u, v).weight=w \in W$ and (u, v, w) represents a cell with u, v, w as coordinates.

GraphLego by design provides three alternative graph parallel abstractions to partition a graph and to enable locality-optimized access to the stored graph using three different levels of granularity: slice, strip and dice. By utilizing different graph parallel abstractions based on memory resource capacity, GraphLego can process big graphs efficiently on a single PC with different resource capacity, by using a unified multi-level graph parallel abstractions based computation framework.

3.1.1 Dice-based Parallel Graph Abstraction

The dice partitioning method partitions a large graph G into dice-based subgraph blocks and store G in dices to balance the parallel computation tasks and maximize the advantage of parallel processing. Concretely, given $G=(V, E, W)$ and its 3D cube $I=(S, D, E, W)$, we first sort the vertices in



(a) Original Graph (b) Dice Partitioning

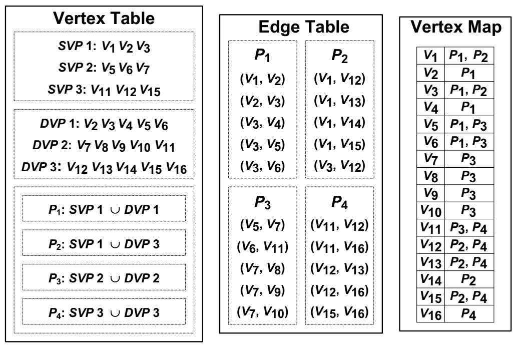
Figure 1: Dice Partitioning: An Example

S and D by the lexical order of their vertex IDs. Then we partition the destination vertices D into q disjoint partitions, called **destination-vertex partitions** (DVPs). Similarly, we partition the source vertices S ($|D|=|V|$) into r disjoint partitions, called **source-vertex partitions** (SVPs). A **dice** 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 one of the SVP, denoting a subset of source vertices, $D_H \subseteq D$ is one of the 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 and its destination vertex from D_H and its edge weight in W_H. Unlike a vertex and its adjacency list (edges), a dice is a subgraph block comprised of 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 u and its edges are typically partitioned into multiple dices. Figure 1 (a) gives an example graph and Figure 1 (b) shows a dice-based partitioning of this example graph. Each of four dice partitions is a dice-based subgraph satisfying the constraint defined by the specific SVP and DVP. Because in-edges and out-edges of a vertex are often applied to different application scenarios, 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. Figure 2 shows the storage organization of the dice partitions in Figure 1 (b), consisting of a vertex table, an edge table (in-edges or out-edges), and a mapping of vertex ID to partition ID.

In GraphLego, dice is the smallest storage unit and by default the original graph G is stored on disk in unordered dices. To provide efficient processing for all types of iterative graph computations, GraphLego stores an original graph using two types of 3D cubes: in-edge cube and out-edge cube, each consists of unordered set of dices of the same type on disk (IEDs or OEDs). This provides efficient access locality for iterative graph computations that require only out-edges or only in-edges or both.

3.1.2 Slice-based Parallel Graph Abstraction

In contrast to dices, slices are the largest partition units in GraphLego. To address the skewed edge weight distribution, we provide the slice-based graph partitioning method, which partitions a 3D cube of graph into p slices along dimension W. p is chosen such that edges with similar weights are clustered into the same partition. Formally, given a 3D cube $I=(S, D, E, W)$, a **slice** of I is denoted 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 W_J. A big advantage of slice partitioning along dimension W is that we can choose those slices that meet the utility requirement



Vertex Table

SVP 1: V_1 V_2 V_3
SVP 2: V_5 V_6 V_7
SVP 3: V_{11} V_{12} V_{15}

DVP 1: V_2 V_3 V_4 V_5 V_6
DVP 2: V_7 V_8 V_9 V_{10} V_{11}
DVP 3: V_{12} V_{13} V_{14} V_{15} V_{16}

P_1: *SVP* 1 $\cup$ *DVP* 1
P_2: *SVP* 1 $\cup$ *DVP* 3
P_3: *SVP* 2 $\cup$ *DVP* 2
P_4: *SVP* 3 $\cup$ *DVP* 3

Edge Table

P_1	P_2
(V_1, V_2)	(V_1, V_{12})
(V_2, V_3)	(V_1, V_{13})
(V_3, V_4)	(V_1, V_{14})
(V_3, V_5)	(V_1, V_{15})
(V_3, V_6)	(V_3, V_{12})

P_3	P_4
(V_5, V_7)	(V_{11}, V_{12})
(V_6, V_{11})	(V_{11}, V_{16})
(V_7, V_8)	(V_{12}, V_{13})
(V_7, V_9)	(V_{12}, V_{16})
(V_7, V_{10})	(V_{15}, V_{16})

Vertex Map

V_1	P_1, P_2
V_2	P_1
V_3	P_1, P_2
V_4	P_1
V_5	P_1, P_3
V_6	P_1, P_3
V_7	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
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 (OEDs)

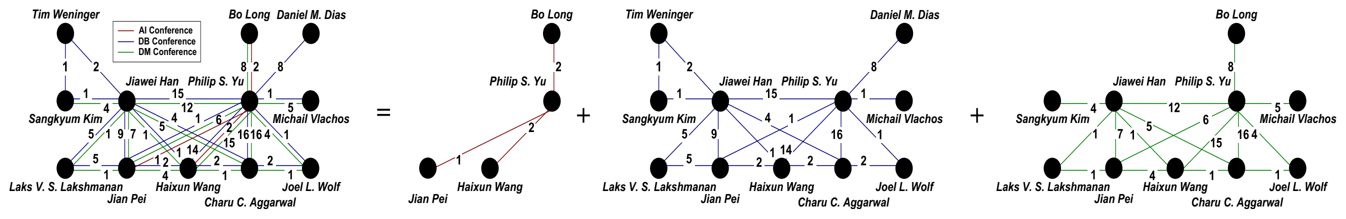
$e.weight$	(0, 0.1]	(0.1, 0.2]	(0.2, 0.3]	(0.3, 0.4]	(0.4, 0.5]	
$edges(\%)$	59.90	18.79	6.99	6.61	5.14	
$e.weight$	(0.5, 0.6]	(0.6, 0.7]	(0.7, 0.8]	(0.8, 0.9]	(0.9, 1.0)	1.0
$edges(\%)$	0.14	0.21	0.08	0.02	0.00	2.12

Table 1: Transition Distribution on DBLPS

to carry out the iterative graph computation according to application-dependent accuracy requirements.

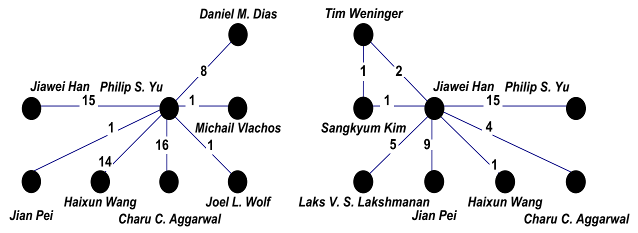
An intuitive example for utilizing slice partitioning is to handle multigraphs. A multigraph is a graph that allows for parallel edges (multiple edges) between a pair of vertices. RDF graph is a typical example of multigraph, where a pair of subject and object vertices may have multiple edges with each annotated by one predicate. Similarly, the D-BLP coauthor graph can also be generated as a coauthor multigraph in terms of 24 computer research fields [46]: AI, AIGO, ARC, BIO, CV, DB, DIST, DM, EDU, GRP, HCI, IR, ML, MUL, NLP, NW, OS, PL, RT, SC, SE, SEC, SIM, WWW. A pair of coauthors in this multigraph can have up to 24 parallel edges, each weighted by the number of co-authored papers in one of the 24 computer science fields [27]. Figure 3 shows an illustrative example of slices. Consider the example co-author graph in Figure 3 (a) with three types of edges: AI, DB and DM, representing the number of coauthored publications on 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 in Figure 3 (b), (c) and (d) respectively, one for each category. If we want to compute the coauthor based social influence among researchers in DB and DM area, we only need to perform iterative computation on the coauthor graph using joint publications in DB and DM conferences and journals.

Another innovative and rewarding usage of slice partitioning is to speed up the iterative graph algorithms on single graphs by performing parallel computation on p slices in parallel. Consider PageRank as an example, the edge weights in the original simple graph are normalized as probabilities in the transition matrix $\mathbf{M}$. Thus, the domain of W is defined on a continuous space over the range $[0, 1]$. In each iteration, PageRank updates the ranking vector $\mathbf{R}$ by iteratively calculating the multiplication between $\mathbf{M}$ and $\mathbf{R}$. However, the transition matrix $\mathbf{M}$ often has skewed edge weight distribution. For instance, the transition matrix for the DBLPS dataset [44] has skewed distribution of transition probabilities (the percentage of edges in the specific weight range to total edges), as shown in Table 1. By introducing dimension W, GraphLego can partition the input DBLPS graph for PageRank into slices along dimension W based on its transition matrix $\mathbf{M}$ and execute the iterative computations at the slice level in parallel. This enables GraphLego to



(a) Coauthor Multigraph (b) AI Slice (c) DB Slice (d) DM Slice

Figure 3: Slice Partitioning: An Example from DBLP



(a) Strip 1 (b) Strip 2

Figure 4: Strip Partitioning of DB Slice

address skewed edge weight distribution much more efficiently as demonstrated in our experiments reported in Section 4, where we set $p = 13, 10, 7, 7, 4$ and 4 for Yahoo, uk-union, uk-2007-05, Twitter, Facebook and DBLPS, respectively. We show that the iterative graph computation using the slice partitioning significantly improves the performance of all graph algorithms we have tested.

The slice partitioning can be viewed as an edge partitioning through clustering by the edge weights. However, when a slice-based subgraph (partition block) together with its intermediate results are too big to fit into the working memory, we further cut the graph into smaller graph parallel units such as dices (see Section 3.1.1) or strips (see Section 3.1.3).

3.1.3 Strip-based Parallel Graph Abstraction

For graphs that are sparse with skewed degree distribution, high degree vertices can incur acute imbalance in parallel computation, and lead to serious performance degradation. This is because the worker thread assigned to a high degree vertex takes much longer time to compute than the parallel threads computing on low degree vertices. To maximize the performance of parallel computation, and ensure better resource utilization and better work balance, in GraphLego we introduce the strip-based graph partitioning, which cuts a graph along either its source dimension or its destination dimension to obtain strips. Compared to the dice-based partitioning that cuts a graph (or slices of a graph) along both source and destination dimensions, strips represent larger partition units than dices. A strip can be viewed as a sequence of dices stored physically together. Similarity, we further cut a slice into strips when single slice can not fit into the working memory.

An efficient way to construct in-edge strips (out-edge strips) is to cut an in-edge cube or slice (out-edge cube or slice) along destination (source) dimension D (S). By cutting an in-edge slice of G, $J{=}(S, D, E_J, W_J)$, along D into q in-edge strips, each strip is denoted as $K{=}(S, D_K, E_K, W_J)$, where $D_K \subseteq D$ is a subset of destination vertices, 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 and it has all OEDs of a SVP. Figure 4 gives an illustrative example of strip-based partitioning, where two strips are extracted from the DB slice in Figure 3 (c), i.e., all coauthored DB links of *Daniel M. Dias*, *Michail Vlachos* and *Philip S. Yu*, and all coauthored DB links of *Jiawei Han*, *Sangkyum Kim* and *Tim Weninger*. Another important feature of our graph partitioning methods is to choose smaller subgraph blocks such as dice partition or strip partition to balance the parallel computation efficiency among partition blocks and to use larger subgraph blocks such as slice partition or strip partition to maximize sequential access and minimize random access.

3.1.4 Graph Partitioning Algorithms

In the first prototype of GraphLego, we implement the three parallel graph abstraction based partitioning, placement and access algorithm using a unified graph processing framework with the top-down partitioning strategy. Given a PC, a graph dataset and a graph application, such as PageRank or SSSP, GraphLego provides the system default partitioning settings on p (*#Slices*), q (*#Strips*) and r (*#Dices*). We defer the detailed discussion on the settings of optimal partitioning parameters to Section 3.5. Based on the system-supplied default settings of the partitioning parameters, we first partition a graph into p slices, and then we partition each slice into q strips, and partition each strip into r dices. The partitioning parameters are chosen such that each graph partition block and its intermediate results will fit into the working memory. In GraphLego, we first partition the source vertices of a graph into SVPs, partition destination vertices of the graph into DVPs and then partition the graph into edge partitions slice by slice, strip by strip or dice by dice. Figure 2 gives an example of vertex partitioning and edge partitioning for the graph in Figure 1. Based on the system-recommended partitioning parameter, we store the graph in the physical storage using the smallest partition unit given by the system configuration. Dice is the smallest and most frequently used partition block in GraphLego (see Table 6).

GraphLego provides a three-level partition-index structure to access dice partition blocks on disk slice by slice, strip by strip or dice by dice. When the graph application has sufficient memory to host the entire index in memory, sequential access to dices stored in physical storage can be maximized. For example, PageRank algorithm requires computing the ranking score for every vertex using its incoming edges and its corresponding source vertices in each iteration. Thus, the PageRank implementation in GraphLego will start to access the graph by one in-edge strip at a time. For each strip, we check if there are multiple slices corresponding to this strip, For each strip and a corresponding slice, we access the dices corresponding to the strip and the slice. GraphLego provides a function library to

Algorithm 1 **GraphPartitioning**$(G, app, p, q, r, flag)$

```
1: Sort source vertices by the lexical order of vertex IDs;
2: Sort destination vertices by the lexical order of vertex IDs;
3: Sort edges by source vertex ID, destination vertex ID and edge
   weight;
4: switch($flag$)
5:    case 0: select $p, q, r$ input by user;
6:    case 1: detect resource, calculate $p, q, r$ for $app$ online;
7:    case 2: select the optimal $p, q, r$ with offline learning;
8: Divide $W$into $p$ intervals;
9: Split $G$ into $p$ in-edge slices;
10: Divide $D$ into $p$ DVPs; //partition destination vertices
11: Split $p$ in-edge slices into $p \times q$ in-edge strips; //partition edges
    in each in-edge slice into q strip-based edge partitions
12: divide $S$ into $q$ SVPs; //partition source vertices
13: Split $p \times q$ in-edge strips into $p \times q \times r$ IEDs; //partition edges in
    each in-edge strip into r dice-based edge partitions
14: Compress DVPs, SVPs and IEDs, and write them back to disk;
    //compress vertex partitions and edge partitions
15: Build the indices for $p$ slices, $p \times q$ strips and $p \times q \times r$ IEDs;
```

support various iterative graph applications with a conventional vertex-oriented programming model. Algorithm 1 provides the pseudo code for an example function *GraphPartitioning*. Given a graph G and a graph application, say PageRank, it constructs an in-edge cube representation of G and partitions its in-edge cube through slice, strip and dice abstractions. We do not build the out-edge cube of graph since PageRank does not use outgoing edges.

3.2 Access Locality Optimization

We describe two access locality based optimizations implemented in the first prototype of GraphLego: (1) graph index structure for indexing slices, strips and dices; and (2) graph partition-level compression for optimizing disk I/Os.

Partition Index. GraphLego stores a graph G in the physical storage as either dices or strips or slices. The iterative graph computation is performed in parallel at the partition level, be it a dice or a strip or a slice. Each partition block corresponds to a subgraph of G. In order to provide fast access to partition blocks of G stored on disk using slice or strip or dice specific conditions, and to ensure that each partition subgraph is loaded into the working memory only once or minimum number of times in each iteration, we design a general graph index structure to enable us to construct the slice index, the strip index or the dice index. The dice index is a dense index that maps a dice ID and its DVP (or SVP) to the chunks on disk where the corresponding dice partition is stored physically. The strip index is a two-level sparse index, which maps a strip ID to the dice index blocks and then map each dice ID to the dice partition chunks in the physical storage. Similarly, the slice index is a three-level sparse index with slice index blocks at the top, strip index blocks at the middle and dice index blocks at the bottom, enabling fast retrieval of dices with a slice-specific condition. In addition, we also maintain a vertex index that maps each vertex to the set of subgraph partitions containing this vertex, as shown in Figure 2. This index allows fast lookup of the partitions relevant to a given vertex.

Partition-level Compression. It is known that iterative computations on large graphs incur non-trivial cost for the I/O processing. For example, the I/O processing of Twitter dataset on a PC with 4 CPU cores and 16GB memory takes 50.2% of the total running time for PageRank (5 iterations). In addition to utilize index, we employ partition-level compression to increase the disk I/O efficiency. Concretely, GraphLego transforms the raw graph data into partition blocks and applies in-memory gzip compression to transform each partition block into a compressed format before storing them on disk. We maintain two buffers in memory, one for input graph data and another for in-memory compressed output graph data. As soon as a sequential read into the input stream buffer is completed, we start the in-memory gzip compression and append the compressed data to the output stream buffer. After finishing the compression, GraphLego sequentially writes the compressed chunks in the output stream buffer to disk. This one-time compression cost at the building time can provide quick access to stored graph partitions and reduce I/O time in each of the graph computation iteration. A gzip-compressed graph file consists of a 2KB head section and multiple partition sections, as shown in Figure 5. The head section stocks the descriptive information about the graph: the number of vertices, the number of edges, the edge type (undirected or directed), the domain of edge weights, the average degree, and the maximum degree. Each partition section consists of a 1KB metadata tag followed by the data in the partition. The tag provides the descriptive information about the specific partition: the data type, the size of partition, the number of edges, the source range, the destination range, the domain of edge weights.

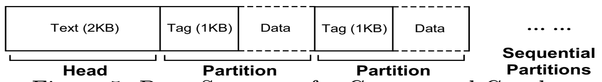


Figure 5: Data Structure for Compressed Graph

3.3 Programmable GraphLego Interface

GraphLego provides a conventional programming interface to enable users to write their iterative graph algorithms using the vertex centric computation model. By supporting the vertex centric programming API, the users of GraphLego only need to provide their iterative algorithms in terms of vertex-level computation using the functions provided in our API, such as *Scatter* and *Gather*. For each iterative graph algorithm defined by users using our API, GraphLego will compile it into a sequence of GraphLego internal function (routine) calls that understand the internal data structures for accessing the graph by subgraph partition blocks. These routines can carry out the iterative computation for the input graph either slice by slice, strip by strip, or dice by dice. For example, PageRank algorithm can be written by simply proving the computation tasks, as shown in Algorithm 2.

For each vertex in a DVP to be updated, GraphLego maintains a temporary local buffer to aggregate received messages. Since each vertex v may contain both in-edges and out-edges, users only need to define two application-level functions to instruct the system-level scatter and gather routines to perform the concrete aggregations. Given a vertex, the *Scatter* function works on a selection of v's neighbors, say the destination vertices of the out-edges of v, to scatter its update based on its vertex state from the previous iteration. Similarly, the *Gather* function works on a selection of v's neighbors, e.g., the source vertices of the in-edges of v, to gather the information in order to update its vertex state, and pass this updated vertex state to the next iteration if the update commits and otherwise assign a partial commit to the gather task.

Vertices are accessed either by DVP or SVP depending on the specific graph algorithm. For each vertex to be updated, GraphLego maintains a temporary local buffer to aggregate

Algorithm 2 PageRank

```
1: Initialize($v$)
2:    $v.rank = 1.0$;
3:
4: Scatter($v$)
5:    $msg = v.rank/v.degree$;
6:    //send $msg$ to destination vertices of $v$'s out-edges
7:
8: Gather($v$)
9:    $state = 0$;
10:   for each $msg$ of $v$
11:   //receive $msg$ from source vertices of $v$'s in-edges
12:      $state$ += $msg$; //summarize partial vertex updates
13:   $v.rank = 0.15+0.85*state$; //produce complete vertex update
```

received messages. We implement the *Scatter* API function as follows: First, an internal routine called *GraphScan* is invoked, which will access the graph dataset on disk by partition blocks (say dices). Then the *PartitionParallel* routine will be invoked to assign multiple threads to process multiple partitions in parallel, one thread per partition subgraph block. For each partition subgraph, the *VertexParallel* routine is called to execute multiple subthreads in parallel, one per vertex. At each vertex thread, the *Scatter* routine is performed. Given that the *Scatter* function at the system level intends to send the vertex state of v to all or a selection of its (out-edge) neighbor vertices, by utilizing the vertex-partition map (see Figure 2), each vertex thread will check if all partition blocks containing v as a source vertex have been processed. If so, then v finishes its scatter task with a commit status; Otherwise, v registers a partial commit.

Similarly, the *Gather* function in our API will be carried out by executing a sequence of four tasks with the first three routines identical to the implementation of the scatter function. The fourth routine is the *Gather* routine, which executes the gather task in two phases: (1) intra-partition gather, performing partial update of vertices or edges within a partition subgraph block, and (2) cross-partition gather, combining partial updates from multiple partition blocks. We call the vertices that belong to more than one subgraph partitions (e.g., dices) the *border* vertices. The cross-partition gather is only necessary for the *border* vertices. The *Gather* routine first records the messages that v has received from the source vertices of v's in-edges in the receive buffer, produces the count of the received messages, combines the update messages in the receive buffer using the local aggregation operation provided in the user-defined gather function, and then store the partial update as the new vertex state of v. At the end of the aggregation operation, if the received message count is the same as the in-degree of v, then we get the final update of v, and store it as the new state of v for this iteration. Otherwise, if the received message count is less than the in-degree of v, we mark this vertex as a *border* vertex, indicating that it belongs to more than one partition blocks and thus needs to enter the cross-partition gather phase. In the next subsection, we will discuss how GraphLego executes the cross-partition gather task to combine the partial updates from different edge partitions at different levels of granularity to generate the complete update.

3.4 Synchronization and Multi-threading

To maximize parallelism in iterative graph computations, GraphLego provides parallel processing at two levels: (1) parallel processing at the subgraph partition level (slice, strip or dice) , and (2) parallel processing at the vertex level A number of design choices are made carefully to support effective synchronization and multi-threading, ensuring vertex and edge update consistency.

Parallel partial vertex update. As vertices in different dice or strip based subgraph partitions belong to different DVPs (or SVPs), and the DVPs (SVPs) from different strip or dice partitions are disjoint within the given graph or a slice of the graph, the vertex update can be executed safely in parallel on multiple strip or dice based partitions, providing partition-level graph parallelism. Furthermore, the vertex update can also be executed safely in parallel on multiple vertices within each strip or dice since each vertex within the same DVP (or SVP) is unique. Thus, GraphLego implements the two-level graph computation parallelism at the partition level and at the vertex level.

However, all the parallel vertex updates at both partition-level and vertex level are partial for two reasons: (1) although the edge sets in different in-edge subgraph partitions are disjoint, an edge may belong to one in-edge partition and one out-edge partition for strip or dice based partitions; thus concurrent edge update needs to be synchronized; (2) a vertex may belong to more than one partitions. Thus, the vertex updates performed within a strip or dice partition are partial and need to do cross-partition gather among strip or dice partitions; and (3) the associated edges of a DVP (or SVP) may lie in multiple slices. Thus, the vertex updates performed concurrently on strip or dice partitions within each slice are partial and need to do cross-partition gather among slices.

For cross-partition gather at slice level, for each vertex, there are at most p partial vertex update states, one per slice. We need to aggregate all the partial vertex update states for each vertex to obtain its final vertex update state before moving to the next iteration. To ensure the correctness and consistency of obtaining the final vertex updates via aggregating such partial vertex update states, during each iteration, GraphLego uses an individual thread to sequentially aggregate all partial vertex updates of a single DVP (or SVP) slice by slice.

Similarly, for cross-partition gather at strip or dice level, GraphLego divides the strip or dice based partitions that share the same DVP (or SVP) into DVP (or SVP) specific partition groups, and sets the number of DVPs (or SVPs) to be processed in parallel by the number of concurrent threads used (*#threads*) such that an individual memory block, i.e., partial update list is assigned to each partition group for cross-partition gather. Now each of the individual threads is dedicated to each edge partition within the DVP (or SVP) specific partition groups and execute one such partition at a time. In order to avoid conflict, GraphLego maintains a *counter* with an initial value of 0 for each specific partition group. When a thread finished the *Scatter* process of an edge partition within the partition group: put the partial update into the partial update list, this thread checks if *counter* is equal to the number of associated edge partitions for the specific partition group. If not, this thread performs *counter*++ and the scheduler assigns an unprocessed edge partition within the same specific partition group to it. Otherwise, we know that GraphLego have finished the processing of all edge partitions within the partition group. Thus, this thread continues to perform the *Gather* process to aggregate all partial updates of this DVP (or SVP) in its partial update list to generate its complete update. Finally, the final update of this DVP (or SVP) in the current iteration are written back to disk. Then, this thread will start to fetch and process the next unfinished or unprocessed DVP (or SVP) and the set of subgraph partitions associated to this DVP (or SVP) in the same manner. We complete one round of iterative computation when vertices in all DVPs (or SVPs) are examined and updated.

Parallel edge update. In GraphLego each edge must belong to one in-edge dice (IED) and one out-edge dice (OED). Thus, edge update is relatively straightforward. GraphLego also implements a two-level parallelism at the strip level by strip threads and at the vertex level by vertex subthreads. An individual strip thread is assigned to a single DVP (or SVP) to sequentially execute the updates of associated edges of this DVP (or SVP) slice by slice. When a DVP (or SVP) thread finishes the updates of all associated edges of a DVP (or SVP), this DVP (or SVP) thread will fetch and process the associated edges of the next unprocessed DVP (or SVP) in the same manner without synchronization.

3.5 Configuration of Partitioning Parameters

Given a total amount of memory available, we need to determine the best settings of the partitioning parameters for achieving the optimal computational performance. GraphLego supports three alternative methods to determine the settings of parameters: user definition, simple estimation, and regression-learning based configurations.

User Definition. We provide user-defined configuration as an option for expert users to modify the system default configuration.

Simple Estimation. The general heuristic used in simple estimation is to determine p (*#Slices*), q (*#Strips*) and r (*#Dices*) based on the estimation of whether each subgraph block for the given partition unit plus the intermediate results will fit into the available working memory. In GraphLego, we provide simple estimation from two dimensions: the past knowledge from regression-based learning and the simple estimation in the absence of prior experiences by estimating the size of the subgraph blocks and the intermediate results depending on the specific graph applications. GraphLego uses the parameter settings produced by simple estimation as the system-defined default configuration.

In summary, the decision of whether to use slice, strip or dice as the partition unit to access the graph data on disk and to process the graph data in memory should be based on achieving a good balance between the following two criteria: (1) We need to choose the partition unit that can best balance the parallel computation workloads with bounded working memory; and (2) we need to minimize excessive disk I/O cost by maximizing sequential disk access in each iteration of the graph algorithm.

Regression-based Learning. A number of factors may impact the performance of GraphLego, such as concrete applications, graph datasets, the number of CPU cores, the DRAM capacity. Thus, for a given graph application, a given dataset and a given server, we want to find the latent relationship between the number of partitions and the runtime. In order to learn the best settings of these partitioning parameters, we first utilize multiple polynomial regression [47] to model the nonlinear relationship between independent variables p, q or r and dependent variable T (the runtime) as an n^{th} order polynomial. A regression model 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

Application	Propagation	Core Computation
PageRank [21]	single graph	matrix-vector
SpMV [22]	single graph	matrix-vector
Connected Components [23]	single graph	graph traversal
Diffusion Kernel [24]	two graphs	matrix-matrix
Inc-Cluster [25]	two graphs	matrix-matrix
Matrix Multiplication	two graphs	matrix-matrix
LMF [26]	multigraph	matrix-vector
AEClass [27]	multigraph	matrix-vector

Table 2: Graph Applications

Graph	Type	#Vertices	#Edges	AvgDeg	MaxIn	MaxOut
Yahoo [40]	directed	1.4B	6.6B	4.7	7.6M	2.5K
uk-union [41]	directed	133.6M	5.5B	41.22	6.4M	22.4K
uk-2007-05 [41]	directed	105.9M	3.7B	35.31	975.4K	15.4K
Twitter [42]	directed	41.7M	1.5B	35.25	770.1K	3.0M
Facebook [43]	undirected	5.2M	47.2M	18.04	1.1K	1.1K
DBLPS [44]	undirected	1.3M	32.0M	40.67	1.7K	1.7K
DBLPM [44]	undirected	0.96M	10.1M	21.12	1.0K	1.0K
Last.fm [45]	undirected	2.5M	42.8M	34.23	33.2K	33.2K

Table 3: Experiment Datasets

n_p, n_q and n_r are the highest orders of variables p, q or r, and ϵ represents the error term of the model.

We then select m samples of (p_l, q_l, r_l, T_l) $(1 \le l \le m)$ from the existing experiment results, such as the points in Figure 15 (c)-(d), to generate the following m linear equations:

$$\begin{aligned} T_1 &= \sum_{i=1}^{n_p}\sum_{j=1}^{n_q}\sum_{k=1}^{n_r}\alpha_{ijk}p_1^i q_1^j r_1^k + \epsilon \\ &\cdots \quad \cdots \\ T_m &= \sum_{i=1}^{n_p}\sum_{j=1}^{n_q}\sum_{k=1}^{n_r}\alpha_{ijk}p_m^i q_m^j r_m^k + \epsilon \end{aligned} \tag{1}$$

We adopt the least squares approach [48] to solve the above overdetermined linear equations and generate the regression coefficients α_{ijk}. Finally, we utilize a successive convex approximation method (SCA) [49] to solve this polynomial programming problem with the objective of minimizing the predicted runtime and generate the optimal p, q and r. The experimental evaluation demonstrates that our regression-based learning method can select the optimal setting for the partitioning parameters, which gives GraphLego the best performance under the available system resource.

4. EXPERIMENTAL EVALUATION

We evaluate the performance of GraphLego using a set of well-known iterative graph applications in Table 2 on a set of large real world graphs in Table 3. DBLPS is a single heterogeneous graph with three kinds of vertices: 964,166 authors, 6,992 conferences and 363,352 keywords, and 31,962,786 heterogeneous links. By following the edge classification method in [27], we construct a coauthor multigraph, DBLPM, with 964,166 authors and 10,180,035 coauthor links. Each pair of authors have at most 24 parallel coauthor links, each represents one of the 24 computer research fields [46], as mentioned in Section 3.1.2. Similarly, we build a friendship multigraph of Last.fm with 2,500,000 highly active users and 42,782,231 parallel friendship links. All the 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 GraphLego with three existing representative single PC systems: **GraphLab** [4], **GraphChi** [11] and **X-Stream** [16].

We use regression-based learning described in Section 3.5 to choose the optimal setting for the GraphLego partitioning

Graph	GraphLab	GraphChi	X-Stream	GraphLego
Yahoo [40]	0	6073	28707	15343
uk-union [41]	0	4459	20729	10589
uk-2007-05 [41]	0	2826	13965	5998
Twitter [42]	6210	1105	5620	2066
Facebook [44]	31	4	25	9
DBLPS [43]	23	3	18	7

Table 4: Building Time (seconds)

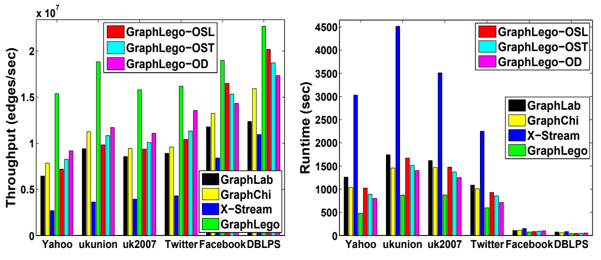
parameters p, q and r_i or r_o. To show the significance of multi-level graph parallel abstraction powered by resource-adaptive customization, we evaluate and compare the following four versions of GraphLego: (1) **GraphLego** with the optimal setting of p, q and r; (2) **GraphLego-OSL** with only the optimal p ($\#Slices$); (3) **GraphLego-OST** with only the optimal q ($\#Strips$); and (4) **GraphLego-OD** with only the optimal r ($\#Dices$).

4.1 Preprocessing Efficiency

Most of graph processing systems transform the raw graph text files into their internal graph representations through preprocessing. The preprocessing step typically performs three tasks: (1) read a raw graph dataset into memory block by block, (2) transform raw data blocks into the system-specific internal storage format, and (3) write the preprocessed data back to disk block by block. Table 4 compares the preprocessing time by four graph parallel systems. GraphLab can directly work on those raw graph text files with ordered edges, such as Yahoo, uk-union and uk-2007-05. Thus there is no building time for these datasets. However, for graph datasets with unordered edges, GraphLab needs to presort the graph text files. In this case, GraphLab tends to be the slowest in building time. The building time in X-Stream is usually 2-3 times slower than GraphLego since X-Stream uses a single thread to execute the import task line by line. GraphChi utilizes multiple threads to perform the transformation task and it only needs to build one copy of in-edges. GraphLego imports data faster than X-Stream and GraphLab, but slower than GraphChi, primarily due to the fact that GraphLego builds two copies of the raw graph: in-edge cube and out-edge cube and GraphLego executes in-memory gzip compression in parallel to reduce the total I/O cost for iterative graph computations.

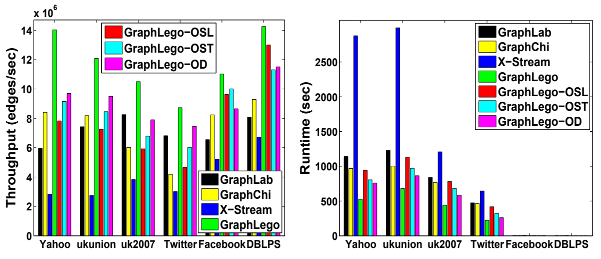
4.2 Execution Efficiency on Single Graph

Figures 6-8 present the performance comparison of iterative algorithms on a single graph with different graph-parallel implementations. Figure 6 (a) shows the throughput ((#edges processed per second)) comparison of PageRank on six real graphs with different scales: Yahoo, uk-union, uk-2007-05, Twitter, Facebook and DBLPS with $\#iterations= 1, 2, 3, 5, 40, 30$ respectively. GraphLego (with the optimal numbers of slices, strips and dices) achieves the highest throughput (around 1.53×10^7-2.27×10^7) and consistently higher than GraphLab, GraphChi and X-Stream and outperforms all versions of GraphLego with partial optimization on all six datasets, with the throughput values by GraphLego-OSL as the lowest (9.38×10^6-2.20×10^7), especially on the largest three graphs. Comparing among the three partial-optimization versions of GraphLego, GraphLego-OSL achieves the highest throughput on two smaller datasets (Facebook and DBLPS) but GraphLego-OD obtains the best performance on other four large-scale graphs (Facebook and DBLPS). This demonstrates that, when handling large-scale graphs, we should focus more on the optimization of r_i (or r_o) by drilling down to the lowest level of granularity in multi-level abstraction to ensure better work balance. On the other hand, the optimization of p should be emphasized for addressing small-scale graphs by rolling up to the highest level of granularity to maximize the sequential bandwidth.



(a) Throughput (b) Runtime

Figure 6: PageRank on Six Real Graphs



(a) Throughput (b) Runtime

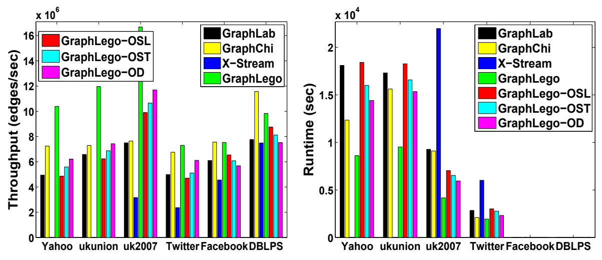
Figure 7: SpMV on Six Real Graphs

Figure 6 (b) compares the running time by different graph-parallel models, including the loading time of preprocessed graph partitions from disk partition by partition, the in-memory decompression time (only for GraphLego and its variants), the computation time, and the time to write results back to disk. The runtime comparison is consistent with the throughput evaluation in Figure 6 (a). The GraphLego family outperforms GraphLab, GraphChi and X-Stream in all experiments. X-Stream achieves the worst performance on all six graph datasets. Although both GraphChi and GraphLab are slower than all versions of GraphLego, GraphChi is relatively faster than GraphLab, as it breaks the edges of large graph into small shards and sort edges in each shard for fast access.

Similar trends are observed for the performance comparison of SpMV and Connected Components (CC) in Figure 7 and Figure 8 respectively. Given that X-Stream failed to work on Yahoo and uk-union when running up to 36,000 seconds in the experiment of CC, we did not plot X-Stream for these two datasets in Figure 8. Compared to GraphLab, GraphChi and X-Stream, GraphLego (with the optimization of all parameters) doubles the throughput and runs twice faster in seconds.

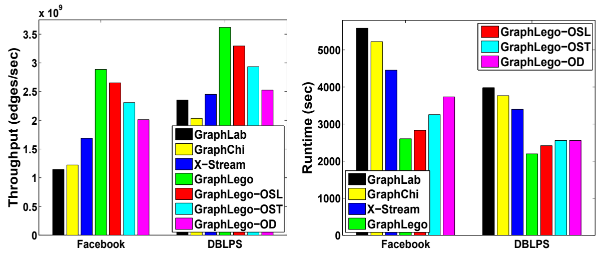
4.3 Execution Efficiency on Multiple Graphs

Figures 9-11 present 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 and graph traversal ($O(n^2)$), we only compare the performance by different graph-parallel models on two s-



(a) Throughput (b) Runtime

Figure 8: Connected Components on Six Real Graphs



(a) Throughput (b) Runtime

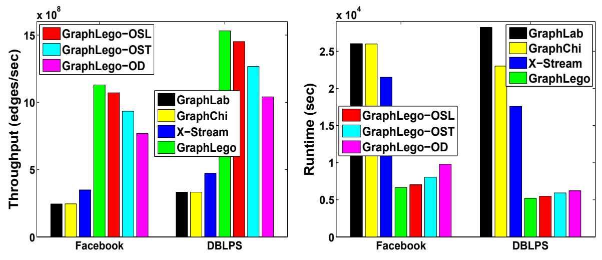
Figure 9: Matrix Multiplication on Two Real Graphs

maller datasets: Facebook and DBLPS. We observe the very similar trends as those shown in Figures 6-8. All versions of GraphLego significantly prevail over GraphLab, GraphChi and X-Stream in all efficiency tests, and GraphLego (full optimization of partitioning parameters) obtains the highest throughput.

4.4 Effect of Partitioning on Dimension W

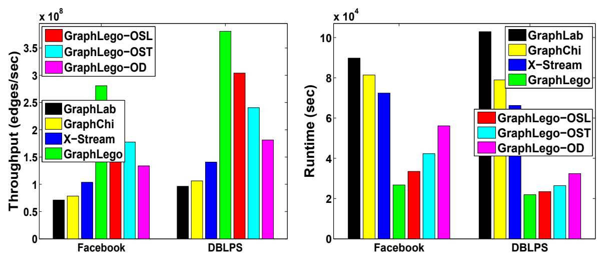
Figure 12 exhibits the efficiency comparison of PageRank on multiple graphs by GraphLego with different numbers of slices along dimension W. Although the above graphs are simple graphs (non-multigraphs) with the edge weights of 0 or 1, PageRank [21] needs to iteratively calculate the multiplication between the transition matrix M and the ranking vector R. Similar examples include Diffusion Kernel [24] which repeatedly computes the power of the generator H (real symmetric matrix), and Inc-Cluster [25] which iteratively calculates the power of the transition matrix P_A. To reduce the repeated cost of calculating the transition probabilities, instead of stocking the original simple graphs, we store the above graphs with their representations of transition matrix for PageRank in our current implementation. From the runtime curve (or the throughput curve) for each dataset, we have observed that the optimal #Slices on different graphs are quite different, depending on the graph size and the edge weight distribution: large-scale graphs contain more vertices and edges such that there exists more distinct edge weights. The optimal values of #Slices on Yahoo, uk-union, uk-2007-05, Twitter, Facebook and DBLPS are 13, 10, 7, 7, 4 and 4, respectively.

Figures 13-14 present the performance comparison of two multigraph algorithms with the GraphLego implementation. LMF [26] is a graph clustering algorithm based on multigraph in both unsupervised and semi-supervised settings, with the Linked Matrix Factorization (LMF) to extract reliable features and yield better clustering results. AEClass [27] transforms the problem of multi-label classification of heterogeneous information networks into the task of multi-label classification of coauthor (or friendship) multigraph based on activity-based edge classification. GraphLego



(a) Throughput (b) Runtime

Figure 10: Diffusion Kernel on Two Real Graphs



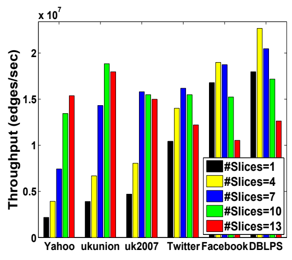
(a) Throughput (b) Runtime

Figure 11: Inc-Cluster on Two Real Graphs

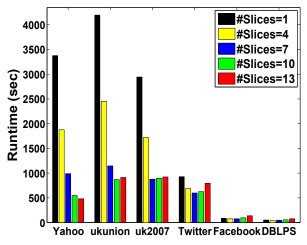
partitions the 3D cube of a multigraph into p slices along dimension W. Each slice consists of parallel edges with a unique semantics. By hashing the parallel edges with the same semantics into the same partition, each slice corresponds to one partition, and represents a subgraph with only those edges that have the corresponding semantics included in the hash bucket for that partition. It is observed that the running time is relatively long when we ignore the slice partitioning, i.e, #Slices = 1. This demonstrates that GraphLego can deliver significant speedup for big multigraph analysis by running iterative computations on p slices with different semantics in parallel. In addition, we have observed that the optimal value of #Slices for a specific multigraph is related to the graph size, the edge weight distribution, and the graph algorithm.

4.5 Decision of #Partitions

Figure 15 measures the performance impact of different numbers of partition on GraphLego with PageRank running over Twitter, Diffusion Kernel and Inc-Cluster running on Facebook. The x-axis shows different settings of the number of partition units (slices, strips and dices). We vary the number of one partition unit, say slice, from 1 to 10,000 and fix the settings of other two units, say setting strips 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 partition unit, 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 the bigger units often lead to substantial work imbalance in iterative graph applications. On the other hand, the smaller units may result in frequent external storage access and lots of page replacements between units lying in different pages. Among three partition units, the policy of dice partition achieves the best performance on large-scale graphs but the policy of slice partition achieves the best performance on small-scale graphs.

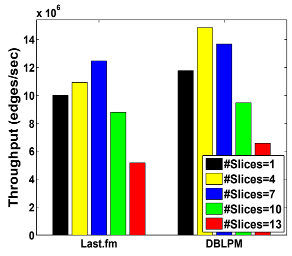


(a) Throughput

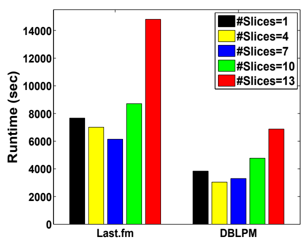


(b) Runtime

Figure 12: PageRank: Vary *#Slices* on Dimension *W*



(a) Throughput

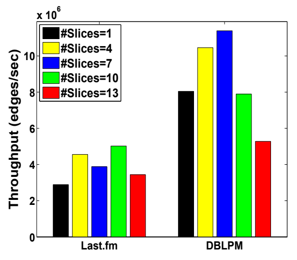


(b) Runtime

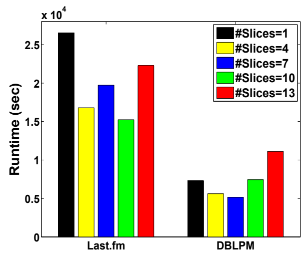
Figure 13: LMF: Vary *#Slices* on Dimension *W*

Figures 15 (e)-(f) measures the CPU utilization by GraphLego for three real applications. The CPU utilization rate on each application increases quickly when the number of partitions (#Slices, #Strips, or #Dices) is increasing. This is because for the same graph, the larger number of partitions gives the smaller size per partition and the smaller partition units in big graphs often lead to better workload balancing when graph computations are executed in parallel. Figures 15 (g)-(h) show the memory utilization comparison. The memory utilization for each application is totally contrary to its CPU utilization when the number of partition units is increasing: the smaller the number of partitions, the larger size each partition will have, thus the larger the memory usage.

Figure 16 shows the effectiveness of the predicated runtime with regression-based learning method for PageRank over Twitter, Diffusion Kernel and Inc-Cluster on Facebook. Instead of the biased or incorrect decision made with experiential knowledge, GraphLego utilizes the multiple polynomial regression model and the successive convex approximation method to discover the optimal numbers of partition units to minimize the execution time. In spite of the common fixed partition scheme, GraphLego implements the 3D cube storage of graph and multi-level graph parallel abstraction to support access locality for various graph analysis algorithms, graphs with different characteristics, and PCs with diverse configurations by drilling down to the lowest level of granularity or by rolling up to the highest level of granularity in multi-level abstraction. The predication curve fits very well with the real execution curve on two settings of the optimal number of partition units (slices and dices) respectively, especially in data points corresponding to the optimal runtime (#Slices=75-250 and #Dices=50-500). Compared to the computational cost of iterative graph applications, the prediction cost is very small due to very small $n_p, n_q, n_{r_i}, n_{r_o} \ll |V|$ for large-scale graphs. In the current implementation, we set $n_p{=}n_q{=}n_{r_i}{=}n_{r_o}{=}3$ in Eq.(1). For the experiment of PageRank on Twitter by GraphLego in Figures 6, the computation time is 599 seconds but the prediction time is only 12 seconds.

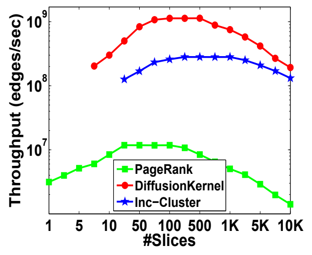


(a) Throughput

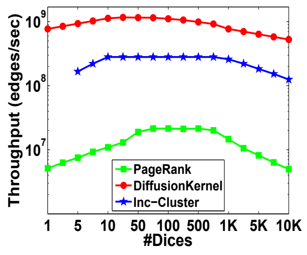


(b) Runtime

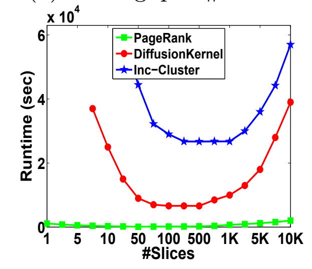
Figure 14: AEClass: Vary *#Slices* on Dimension *W*



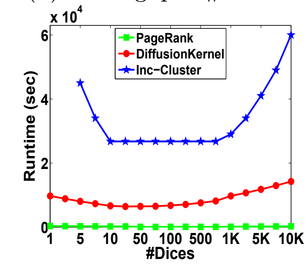
(a) Throughput:#Slices



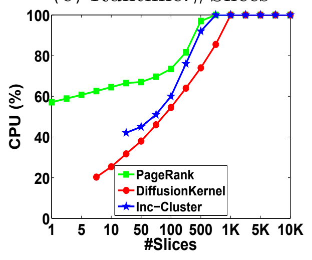
(b) Throughput:#Dices



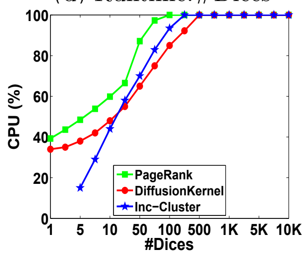
(c) Runtime:#Slices



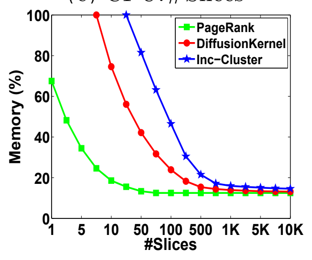
(d) Runtime:#Dices



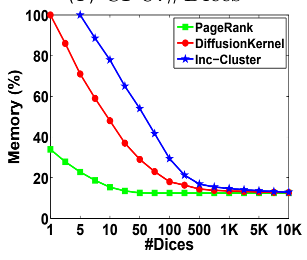
(e) CPU:#Slices



(f) CPU:#Dices



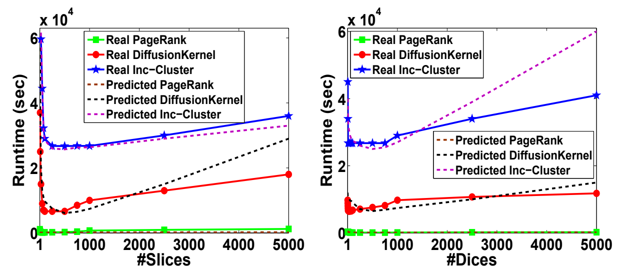
(g) Memory:#Slices



(h) Memory:#Dices

Figure 15: Impact of #Partitions

Tables 5 and 6 compare the optimal p (*#Slices*), q (*#Strips*), r (*#Dices*) generated by offline regression-based learning. Table 5 exhibits the optimal parameters by PageRank on three datasets using two PCs with different memory capabilities. From Table 5, we observe that the multiplications of $p * q * r$ on the 2-core PC are about 2.67-5.97 times than the multiplications of $p * q * r$ on the 4-core PC. This indicates that GraphLego can achieve good performance based on a simple estimation in terms of the optimal parameter setup



(a) Different #Slices (b) Different #Dices

Figure 16: Runtime Prediction

	PC (16 GB memory)			PC (2 GB memory)		
Dataset	Facebook	Twitter	Yahoo	Facebook	Twitter	Yahoo
p (*#Slices*)	4	7	13	4	8	9
q (*#Strips*)	3	5	4	4	10	12
r (*#Dices*)	0	4	8	2	7	23

Table 5: Optimal Partitioning Parameters for PageRank

on the existing machines as long as the simple estimation lies in the near-horizontal interval of the runtime "U" curve shown in Figure 15. Table 6 compares the optimal parameters recommended by regression-based learning for two applications over three different datasets on the 4-core PC. Clearly, the number of partitions grows as the graph dataset gets larger.

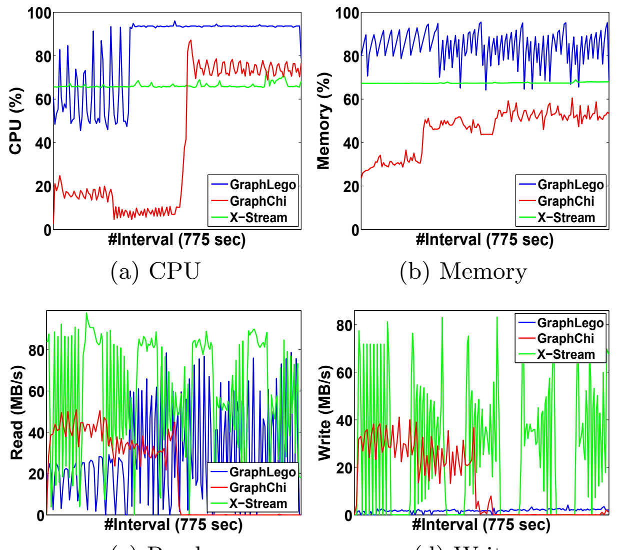
4.6 CPU, Memory and Disk I/O Bandwidth

Figure 17 compares three graph processing systems on CPU utilization, memory utilization and disk I/O bandwidth by PageRank on Twitter with 5 iterations. As shown in Figures 17 (a)-(b), GraphLego achieves the highest utilization rates in both CPU and memory. GraphChi has lower and stable CPU and memory utilization than X-Stream.

Figures 17 (c)-(d) report the I/O bandwidth comparison by three models. GraphLego incurs very small amount of updates but much larger number of reads compared to GraphChi and X-Stream. The I/O bandwidth curves (both I/O read and I/O write) by GraphChi and X-Stream are consistent with those in the X-Stream paper [16]. X-Stream consumes more I/O bandwidth than GraphChi. We observe that (1) Although Graphchi needs to load both in-edges (the shard itself) and out-edges (one sliding window of each of other shards) of a shard into memory, GraphChi updates this shard in memory and then directly write the update to disk. The size of a sliding window of one shard is much smaller than the size of the shard itself; (2) in X-Stream, the huge graph storage dramatically increases the total I/O cost. In addition, the two-phase implementation also doubles the I/O cost: for each streaming partition, the merged scatter/shuffle phase reads its out-edges from disk and writes its updates to disk, and the gather phase loads its updates from disk. When the graph is relatively dense or the edge update is very frequent at the beginning phase of computational iterations, the number of updates approximately equals to the number of edges. The parallelism of I/O processing may also result in higher disk seek time on the standard PC with a single disk; and (3) the gzip-compressed storage helps GraphLego dramatically reduce the total I/O cost. At each iteration, GraphLego reads the in-edges of each DVP, calculates the updates of vertices in the DVP and writes the updates to the private buffer if the entire vertices can fit into the memory, or writes them back to disk if the working memory is limited. After completing the updating of a DVP,

	PageRank			Connected Components		
Dataset	Facebook	Twitter	Yahoo	Facebook	Twitter	Yahoo
p (*#Slices*)	4	7	13	4	6	8
q (*#Strips*)	3	5	4	2	6	7
r (*#Dices*)	0	4	8	0	4	12

Table 6: Optimal Parameters for PC with 16 GB DRAM



(a) CPU (b) Memory

(c) Read (d) Write

Figure 17: CPU, Memory and Disk I/O bandwidth

GraphLego never needs to read this DVP and its associated edges again until the algorithm enters the next iteration. Thus, GraphLego only has very few or no write I/O costs within each iteration.

5. CONCLUSIONS

We have presented GraphLego, a resource adaptive graph processing system with multi-level graph parallel abstractions. GraphLego has three novel features: (1) we introduce multi-level graph parallel abstraction by partitioning a large graph into subgraphs based on slice, strip and dice partitionings; (2) we dynamically determine the right abstraction level based on the available resource for iterative graph computations. This resource-adaptive graph partitioning approach enables GraphLego to respond to computing platforms with different resources and real-world graphs with different sizes through multi-level graph parallel abstractions; (3) GraphLego uses dice-based data placement algorithm to store a large graph on disk to minimize random disk access and enable more structured in-memory access.

Acknowledgement. This material is based upon work partially supported by the National Science Foundation under Grants IIS-0905493, CNS-1115375, IIP-1230740, and a grant from Intel ISTC on Cloud Computing.

6. REFERENCES

[1] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system - implementation and observations. In *ICDM*, 2009.

[2] 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*, pages 135–146, 2010.

[3] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, 2010.

[4] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. In *UAI*, 2010.

[5] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. Gbase: A scalable and general graph management system. In *KDD*, pages 1091–1099, 2011.

[6] A. Buluc and J. R. Gilbert. The combinatorial blas: Design, implementation, and applications. *IJHPCA*, 25(4):496–509, 2011.

[7] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: Taking the pulse of a fast-changing and connected world. In *EuroSys*, pages 85–98, 2012.

[8] 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. In *PVLDB*, 2012.

[9] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan. Managing large graphs on multi-cores with graph awareness. In *ATC*, 2010.

[10] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[11] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. *OSDI*'12.

[12] Giraph. http://giraph.apache.org/.

[13] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *SIGMOD*, 2013.

[14] 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.

[15] W. Xie, G. Wang, D. Bindel, A. Demers, and J. Gehrke. Fast iterative graph computation with block updates. *PVLDB*, 2013.

[16] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric Graph Processing using Streaming Partitions. In *SOSP*, 2013.

[17] K. Lee and L. Liu. Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. *PVLDB*'13.

[18] Y. Tian, A. Balmin, S. Andreas Corsten, S. Tatikonda, and J. McPherson. From "Think Like a Vertex" to "Think Like a Graph". *PVLDB*, 2013.

[19] 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*, 2014.

[20] P. Yuan, W. Zhang, C. Xie, H. Jin, L. Liu, and K. Lee. Fast Iterative Graph Computation: A Path Centric Approach. In *SC*, 2014.

[21] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *WWW*, 1998.

[22] 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.

[23] X. Zhu and Z. Ghahramani. Learning from Labeled and Unlabeled Data with Label Propagation. In *CMU CALD Tech Report*, 2002.

[24] R. I. Kondor and J. D. Lafferty. Diffusion kernels on graphs and other discrete input spaces. In *ICML*, 2003.

[25] Y. Zhou, H. Cheng, and J. X. Yu. Clustering large attributed graphs: An efficient incremental approach. In *ICDM*, 2010.

[26] W. Tang, Z. Lu, and I. S. Dhillon. Clustering with multiple graphs. In *ICDM*, 2006.

[27] Y. Zhou and L. Liu. Activity-edge centric multi-label classification for mining heterogeneous information networks. In *KDD*, 2014.

[28] 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.

[29] H. Tong, C. Faloutsos, and J.-Y. Pan. Fast random walk with restart and its applications. In *ICDM*, 2006.

[30] H. Ma, H. Yang, M. R. Lyu, and I. King. Mining social networks using heat diffusion processes for marketing candidates selection. In *CIKM*, pages 233–242, 2008.

[31] X. Zhu, Z. Ghahramani, and J. Lafferty. Semi-supervised learning using Gaussian fields and harmonic functions. In *ICML*, 2003.

[32] S. A. Macskassy and F. Provost. A simple relational classifier. In *MRDM*, pages 64–76, 2003.

[33] V. Satuluri and S. Parthasarathy. Scalable graph clustering using stochastic flows: Applications to community discovery. In *KDD*, 2009.

[34] Y. Zhou, H. Cheng, and J. X. Yu. Graph clustering based on structural/attribute similarities. *VLDB*'09.

[35] Y. Zhou and L. Liu. Social influence based clustering of heterogeneous information networks. In *KDD*, 2013.

[36] D. D. Lee and H. S. Seung. Algorithms for non-negative matrix factorization. In *NIPS*, 2000.

[37] Y. Koren. Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *KDD'08*.

[38] H. Ma. An experimental study on implicit social recommendation. In *SIGIR*, pages 73–82, 2013.

[39] Y. Zhou, L. Liu, C.-S. Perng, A. Sailer, I. Silva-Lepe, and Z. Su. Ranking services by service network structure and service attributes. In *ICWS*, 2013.

[40] Yahoo Webscope. Yahoo! AltaVista Web Page Hyperlink Connectivity Graph, circa 2002. http://webscope.sandbox.yahoo.com/.

[41] P. Boldi, M. Santini, and S. Vigna. A Large Time-aware Web Graph. *SIGIR Forum*, 42(2):33–38, 2008.

[42] H. Kwak, C. Lee, H. Park, S. Moon. What is Twitter, a social network or a news media? In *WWW*, 2010.

[43] M. Gjoka, M. Kurant, C. T. Butts, A. Markopoulou. Walking in Facebook: A case study of unbiased sampling of OSNs. In *INFOCOM*, 2010.

[44] http://www.informatik.uni-trier.de/∼ley/db/.

[45] http://www.last.fm/api/.

[46] 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.

[47] J. Cohen, P. Cohen, S. G. West, and L. S. Aiken. *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences*. In *Routledge*, 2002.

[48] O. Bretscher. *Linear Algebra With Applications, 3rd Edition*. In *Prentice Hall*, 1995.

[49] F. Hillier and G. Lieberman. Introduction to Operations Research. In *McGraw-Hill College*, 1995.