A Topology-Aware Framework for Graph Traversals

Jia Meng, Liang Cao, and Huashan Yu(✉)

School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China
{mengjiajia,1300012902,yuhs}@pku.edu.cn

Abstract. Computation on a large-scale graph is to propagate and update the vertex values systematically. Efficient graph computing depends on techniques compatible with the algorithm's value propagating pattern. Graph traversing is a value propagating pattern used by representative graph applications. This paper presents an efficient value propagating framework for large-scale graph traversing applications. By partitioning the input graph based on the topology, it allows values for different source vertices to be propagated together, so as to reduce value propagating overhead. A locality-based vertex partitioning strategy is proposed to improve locality on processors. To improve parallel efficiency of graph traversals, a novel task scheduling mechanism has been devised. The mechanism allows the framework to improve load balance without loss of locality. A prototype for the framework has been implemented. On four large real graphs and a synthetic graph, the work was evaluated with two typical graph applications. By comparing with the owner-computing rule, experimental results show that this work has an overall speedup from 1.28 to 2.67. The speedup to Ligra is more than 5 in most cases.

Keywords: Graph traversing · Graph partitioning · Computation decomposing · Dynamic scheduling · Work stealing

1 Introduction

In the domain of data and network science, information is often linked to form large-scale graphs that may consist of billions of edges. Such a connected data tends to be scale-free that the degree distribution follows a power law, and its effective diameter is also low. The Computation on a connected data is vertex-centric and data-driven. During the computation, values of the vertices are propagated along the edges concurrently, according to value propagating pattern specified by the graph algorithm. For example, breadth-first search (BFS) specifies that every vertex can receive at most one propagated value; and PageRank [3] specifies that a vertex should propagate its newly updated value to every outgoing neighbor. On every vertex, the local value is updated according to the received data, and the new value is propagated in turn except the local value has gotten stable.

Although graph computing technology has been studied extensively in recent years [5, 7, 8, 9–13, 15, 17–21], efficiently processing large-scale graphs remains a grand challenge, due to three factors. First, the computation involves value propagations along billions of edges, resulting in that a large amount of data is accessed randomly and

© Springer International Publishing AG 2017
S. Ibrahim et al. (Eds.): ICA3PP 2017, LNCS 10393, pp. 165–179, 2017.
DOI: 10.1007/978-3-319-65482-9_11

intensively. The locality optimizing strategies vary from one application to another. Second, a vertex and its neighbors are to be updated in an order compatible with the value propagating trace. The parallelism often varies from time to time during the computation. Third, the workload on each vertex depends heavily on both the algorithm's propagating pattern and the input graph's degree distribution. The load balancing strategies are algorithm sensitive and data sensitive. To meet the performance requirements of most large-scale graph applications, a feasible solution is to develop techniques specialized for typical propagating patterns. Each technique optimizes one pattern exhaustively, and enables a class of applications to achieve acceptable performance for most input graphs.

Graph traversing is a typical vertex value propagating pattern. It is used by representative graph applications like BFS, connected-component detection [16], graph-diameter estimation [1, 2, 4, 6, 14], and etc. According to this pattern, values are only allowed to be propagated from visited vertices to those yet unvisited, and a vertex is marked as visited immediately after it has received a propagated value. A graph traversing algorithm starts by propagating values from some source vertex, which is initially marked as visited. Every vertex can receive data from at most one neighbor. The overall value propagating trace is a traversing tree, which covers all vertices reachable from the source. Vertices on the tree are updated systematically. There is often more than one source vertex in applications like connected-component detection and graph-diameter estimation. Different sources can be processed in any order.

This paper presents a parallel graph traversing framework that improves application efficiency with topology-adaptive techniques. The framework elaborately divides vertices of the input graph into a relatively small number of vertex blocks, according to the graph topology and memory distribution of these vertices. Every block is a task scheduling unit during the computation, and is ready for task assignment when at least one of its vertices is ready to be updated. A double-queue task scheduling mechanism has been devised to process these blocks concurrently. This mechanism enables a processor to dynamically select tasks according to both distances from the accessed data and sizes of the tasks, so as to improve both load balance and locality. Furthermore, two strategies are exploited to improve value propagating efficiency. One is to allow every subgraph to select the most appropriate value propagating mechanism. Another is to enable different vertex sources to share value propagating overheads by propagating values simultaneously for these sources. We have implemented a prototype for this framework, and evaluated it with both real and synthetic graphs.

2 Problem Statement and Analysis

In this work, a graph traversing application is represented as a quadruple <V, E, S, *f*>, where V is the input graph's vertex set and E is its edge set, $S \subseteq V$ is the source vertex set, and *f* is the function updating values on every vertex. The graph<V, E> is either directed or undirected. Every edge in E serves as a channel with unlimited bandwidth for exchanging data between the two connected vertices. If <V, E> is undirected, the channels are bidirectional; otherwise, the channels are unidirectional. It always costs

one time unit for an edge to transfer data from its original vertex to the terminal. Every vertex in V has an initial value, and computes a new value for every received value with *f*. When a vertex receives multiple values at the same time, it processes these values independently and simultaneously. On every vertex, the time cost by *f* to process the received values is ignored. The application is to propagate every source's value on the graph non-cyclically. On the propagating trace, every vertex will use *f* to replace the received value with a new one before it is propagated further. Every vertex is allowed to receive at most one value for every source. When multiple values for the same source arrive at the same time, the vertex selects one value randomly.

Obviously, the value-propagating trace for every $s \in S$ is a BFS tree rooted from s on the graph <V, E>. The value-propagating trees for different sources are independent. The application is to construct the value-propagating for every $s \in S$, and perform the required value update with *f* on every reached vertex. Therefore, this application model covers any graph-traversing applications based on BFS.

For the graph traversing application <V, E, S, *f*>, the complexity mainly originates from the intensive and random accesses to the edges and vertices. This section first analyzes the chances for the application to reduce the amount of data accesses. The issues of data access efficiency are discussed later. To be convenient, the following terms are defined.

- **Dot**. A dot is a vertex that has no neighbors, and hence is not reachable by other vertices.
- **Terminal**. If <V, E> is undirected, a terminal is a vertex that has only one neighbor. If <V, E> is directed, a terminal is a vertex that has only either outgoing neighbors or ingoing neighbors.
- **Linear segment**. A linear segment is a path from vertex $v_s \in V$ to $v_e \in V$, where: (a) either v_s or v_e is a terminal; (b) except v_s and v_e, other vertices have exactly two neighbors.
- **Linear path**. A linear path is a path from vertex $v_s \in V$ to $v_e \in V$, where: (a) neither v_s nor v_e is a terminal; (b) both v_s and v_e have more than two neighbors, and other vertices have exactly two neighbors.
- **Netlike graph**. Given a graph <V, E>, its netlike graph consists of all linear paths on it, and is denoted as NG(V, E). A vertex on NG(V, E) is called as a hub vertex when it is also on some linear segments.

2.1 Complexity Analysis

Given a source $s \in S$, the traversing tree often consists of a large amount of vertices, and is constructed by systematically propagating values from previously visited vertices. Initially, s is the only vertex on the traversing tree. The traversing tree is then extended iteratively by propagating values along edges outgoing from its vertices to those unvisited. The newly added vertices are found with either the pushing- or pulling-mechanism. If the pulling-mechanism is selected, every unvisited vertex is a candidate for visiting next, and its ingoing neighbors are inspected one after another. When an ingoing neighbor is found to be visited, the neighbor's value is fetched back and the rest

neighbors are ignored. If the pushing-mechanism is selected, candidates for visiting next are limited to outgoing neighbors of the visited vertices. For every visited vertex, its outgoing neighbors are inspected in some order, and its value is sent to the unvisited ones. Therefore, it is very complex to construct the traversing tree, due to the intensive edge accesses and random vertex accesses.

It is possible for different sources to propagate values together, so as to share the propagating overhead caused by edge accesses and vertex inspections. Let $vg(s_1, s_2)$ be a subgraph where every vertex is reachable to both $s_1 \in S$ and $s_2 \in S$. On $vg(s_1, s_2)$, if it is compatible with the value propagating pattern to start value propagating from the same vertex for both s_1 and s_2, then these two sources can share the same value propagating trace on $vg(s_1, s_2)$. For example, let $vg(s_1, s_2)$ be a linear segment that is reachable to both s_1 and s_2, then s_1 and s_2 must share the same value propagating trace on $vg(s_1, s_2)$. In a typical social network graph like Twitter and Friendster, the linear segments cover more than 15% vertices. When s_1 and s_2 reach NG(V, E) via the same hub vertex, they also can share the same value propagating trace on NG(V, E).

2.2 Efficiency Analysis

As discussed in Sect. 2.1, constructing the traversing trees for different sources simultaneously can make chances for reducing complexity of applications. In this case, values for different sources may reach a vertex via different edges concurrently; and every vertex needs a vector to indicate its visiting statuses. The vector's i-th element designates whether current vertex has been visited from the i-th source. A vertex is said to be active when it is still unvisited for at least one source vertex. A vertex $v \in V$ is de-noted as a frontier when there is at least one source $s \in S$ that: (a) v has been visited by s; (b) it is not sure whether v's every outgoing neighbor has been visited by s.

Given a graph traversing algorithm, its computation on the linear segments is relatively simple. Without loss of generality, we can assume that there is at most one frontier on every linear segment. Hence, the vertex to be updated next is always the frontier's outgoing neighbor. However, the computation on NG(V, E) is much more complex, since every active vertex is a candidate for updating next and the update may require more than one frontier's value. To maximize the parallelism, the computation is divided into a sequence of super-steps. For two reasons, the pulling-mechanism is selected to propagate values from frontiers to their active neighbors. One is that most active vertices on a super-step are to be updated, since the input graph tends to be scale free and neighbors of a few frontiers often cover most vertices. Another is that the pushing-mechanism requires writing to a vertex's neighbors frequently and randomly.

To improve efficiency for the computation on NG(V, E), two key issues are to be addressed for each super-step. One is load balance. Different active vertices can be independently processed. On every active vertex, the computation is to inspect its ingoing neighbors and update local value with these included in current frontiers. Hence the workload on an active vertex tends to be proportional to its degree. Another key issue is locality. The active vertices on each super-step are random, and number of active vertices varies greatly from one super-step to another. The value-propagating efficiency will be significantly cut down if neighbors of a large amount of active vertices are randomly accessed.

3 A Topology-Aware Value-Propagating Framework

This section presents a topology-aware framework to propagate values for graph traversals. We assume that a computer consists of n computing nodes, and each node has m processors and a local memory. The distance from a processor to its local memory is shorter than that to any remote memory. The input graph's vertices are equally divided into n blocks. Each block and edges associated with these vertices are saved in one computing node. It is also assumed that a vertex can buffer an initial value and all the updates on it. When the initial value is propagated, the buffered updates are to be propagated together.

Given a graph traversing application <V, E, S, *f*>, the framework schedules its computation on different vertices carefully with three strategies. The first strategy is to partition the input graph's edges according to the graph topology. It enables the framework to reduce edge accesses by selecting value propagating mechanism for every subgraph independently. It also enables the framework to find the chances for sharing value propagating overhead between different source vertices. The second strategy is to partition the input graph's vertices according to both their memory distribution and graph topology. This strategy enables the framework to efficiently filter out vertices that are unreachable to current frontiers, and to improve locality on each processor. The third strategy is to schedule computation partitions greedily and dynamically. The framework carefully selects a processor for every computation partition, according to both every processor's workload and the partition's data accessing efficiency on different processors.

Before the application starts its graph traversals, the framework first partitions the input graph's edges, resulting in a netlike graph and a set of linear segments. Then it partitions the vertex block on each computing node into a relatively small number of vertex chunks. With these subgraphs and chunks, the framework partitions the computation into three kinds of tasks. Every source on some linear segment represents a type-I task, which is to traverse on the linear segment from the source. Every chunk and current frontiers represents a type-II task, which is to (a) propagate values along edges outgoing from current frontiers to the chunk and (b) update the chunk's vertices with the propagated values. Every updated hub vertex and a linear segment outgoing from the hub represents a type-III task, which is to propagate the hub's new value on the linear segment.

By concurrently executing the computations on different subgraphs, the framework constructs the traversing trees for different source vertices simultaneously. At the beginning, every type-I task is assigned to a processor and is executed independently. The framework will not propagate values on the netlike graph, until all type-I tasks have been completed. Value propagating procedure on the netlike graph is divided into a sequence of super-steps. The computation on each super-step is partitioned into a set of type-II tasks. These type-II tasks are greedily scheduled to improve both data accessing efficiency on each processors and load balance between different processors. After computation on the netlike graph has been completed, the type-III tasks are scheduled to be executed on different processors concurrently.

To reduce edge accesses in constructing the traversing trees, the framework independently selects propagating mechanism for every subgraph. If the subgraph is a linear segment, the pushing mechanism is selected; otherwise, the pulling mechanism is selected.

3.1 A Topology-Based Edge Partitioning Strategy

By partitioning the input graph's edges according to the graph topology, our graph traversing framework partitions the input graph into dots, linear segments and a net-like graph. Dots are ignorable, since they are always unreachable. Based on the net-like graph

Algorithm 1. The three-phased graph traversing algorithm.

Phase I

$\emptyset \Rightarrow ihub(S)$;
for every $s \in S$ that is on some linear segment $ls(s)$ **do**
  propagate value of s on $ls(s)$;
  if v is the hub vertex of $ls(s)$ and is visited **then**
    $\{v\} \cup ihub(S) \Rightarrow ihub(S)$;
  end if
end do

Phase II

$\emptyset \Rightarrow ohub(S)$;
$\emptyset \Rightarrow$ next frontier;
$ihub(S) \cup \{s \in S : s \text{ is on } NG(V,E)\} \Rightarrow$ frontier;
$\{v : v \text{ is on } NG(V,E)\} \Rightarrow$ active vertices;
while neither active vertices nor frontiers is null **do**
  for every $v \in$ active vertices **do**
    find_and_update(v) $\Rightarrow$ flag;
    if flag is true **then**
      $\{v\} \cup$ next frontier $\Rightarrow$ next frontier;
      if v is a hub vertex **then**
        $\{v\} \cup ohub(S) \Rightarrow ohub(S)$;
      end if
    end if
    if v has been reached by every $s \in S$ **then**
      active vertices $- \{v\} \Rightarrow$ active vertices;
    end if
  end for
end while

Phase III

for every linear segment $ls(v)$ that is reachable from $v \in ohub(S)$ **do**
  propagate value of v on $ls(v)$;
end do

and linear segments, computation on the input graph can be described with a three-phased algorithm. The algorithm constructs the traversing trees for different source vertices simultaneously, and enables different sources to share value propagating overhead automatically.

Algorithm 1 is the three-phased algorithm. The first phase of the three-phased algorithm is to propagate values on linear segments with the pushing mechanism. Computation of this phase is decomposed into a set of independent type-I tasks. Every task processes one source that is on some linear segment, and creates the source's value propagating trace on the linear segment. If a hub vertex is visited by some type-I task, then the hub is included in *ihub*(S). The second phase is to propagate values on NG(V, E) with the pulling mechanism, consisting of a sequence of super-steps. Sources on NG(V, E) and vertices in *ihub*(S) are the first super-step's frontiers. On each super-step, every active vertex independently tries to update its value by finding its ingoing neighbors from current frontiers and comparing its visiting status with those found. A vertex is a frontier of the next super-step when its value is updated. If a hub vertex is updated, then the hub is included in *ohub*(S). The last phase is to propagate values on linear segments outgoing from hub vertices in *ohub*(S). The computation consists of a set of independent type-III tasks. For every line segment that is reachable to $h_v \in ohub(S)$, a type-III task is executed. The task propagates value of h_v on the line segment, using the pushing mechanism.

3.2 A Locality-Based Vertex Partitioning Strategy

This strategy aims at improving locality on each processor when vertex values are propagated with the pulling-mechanism. The vertices on each computing node are partitioned into a relatively small number of vertex chunks carefully, according to both their memory distribution and graph topology. Every chunk consists of a set of vertices that are continuously saved. It can be used to represent a type-II task, which is to propagate values from current frontiers to vertices included the chunk. And the task's workload is estimated with the graph's topology information. Different chunks can be processed concurrently to make use of parallel processors.

We estimate the workload on vertex v with Eq. (1), where c_s is a constant denoting the overhead for inspecting the vertex's visiting status, c_e is a constant denoting the overhead for inspecting one ingoing neighbor. If the input graph is undirected, *ideg*(v) is the vertex's degree; otherwise, *ideg*(v) is either the ingoing degree or outgoing degree, depending on the value propagating direction. When there are enough chunks with carefully selected upper bound workload, dynamic task scheduling mechanism can be used to improve load balance without loss of locality.

$$\text{workload(v)} = c_s + ideg(\text{v}) \times c_e \tag{1}$$

In our topology-aware framework, the pulling-mechanism is used on the netlike graph only. When all vertices in a chunk are outside of the netlike graph, the chunk is denoted as a *vacancy*, and can be filtered out for computation on the netlike graph. To enable this kind of vertex filtering and to make full use of parallel processors, the framework divides vertices on every computing node into about $m \times \alpha$ non-vacancy chunks

independently, where m is processor number of the computing node and α is an experimental constant. Let *niv* be the number of local vertices that are outside of the netlike graph. We first search the vacancies, and each one must consist of at least $niv \div (m \times \alpha)$ vertices. These vacancies divide the rest vertices into a set of initial non-vacancy chunks. Let *twb* be the total workload of these initial non-vacancy chunks. Each initial non-vacancy chunk is further divided into a minimum of chunks, where each one's workload should be no more than $twb \div (m \times \alpha)$.

3.3 A Double-Queue Task Scheduling Strategy

As discussed above, computation on the netlike graph is divided into a sequence of super-steps, and each super-step consists of a set of independent type-II tasks. Every type-II task processes one non-vacancy chunk. To improve each super-step's parallel efficiency, we have devised a two-level queue to schedule its tasks dynamically. This task scheduling mechanism synthesizes three typical task scheduling techniques. The first is the owner-computing rule, aiming at improving locality on processors. The second is the dynamic task scheduling technique, aiming at improving load balance between processors in a computing node. The last is the work stealing technique, aiming at improving load balance between computing nodes.

Our topology-aware framework maintains two queues on every computing node. Let *cpn* be a computing node, **tque**(*cpn*) and **dque**(*cpn*) denote these two local queues separately. The **tque**(*cpn*) consists of non-vacancy vertex chunks local to *cpn*. It automatically computes average workload of these chunks, and sorts them according to both their workloads and memory addresses. A chunk is denoted as heavy if its workload is greater than the average; otherwise, it is denoted as light. A heavy chunk is in front of any light chunk, so as to enable load balance between processors. Chunks of the same kind are further sorted according to their memory addresses, so as to improve locality on processors. Every element in **dque**(*cpn*) represents one computing node. It sorts the elements according distances from local processors to the corresponding computing nodes. Local memory of **dque**(*cpn*)[*i*] is not farer way from *cpn*'s local processors than that of **dque**(*cpn*)[$i + 1$].

When a super-step is executed, each computing node schedules tasks in its local **tque** independently, and complex tasks are first assigned to processors. Different tasks are executed concurrently. When a processor is free, it submits first submits a task apply to **dque**(*cpn*)[0], attempting to get a task from **tque**(*cpn*). If the processor fails in getting tasks from **dque**(*cpn*)[*i*], it then submits task applies to **dque**(*cpn*)[$i + 1$], trying to steal tasks from the remote computing node specified by **dque**(*cpn*)[$i + 1$].

4 Implementation

We have developed a prototype with C/C++ for the topology-aware value propagating framework. The prototype is for NUMA architecture and uses Pthreads to execute parallel computations. The prototype consists of an edge slicer, a vertex slicer and a value propagator (Fig. 1). The edge slicer partitions a graph's edges according to the

topology, and divides the graph into a netlike graph and a set of linear segments. The vertex slicer partitions a graph's vertices according to both their memory distribution and graph topology. One each computing node, it divides the local vertices into a set of vacancies and non-vacancy chunks. The vertex slicer sets α required by the locality-based computation decomposing mechanism to be 16. The value propagator automatically propagates values on a graph traversing application's input graph, and calls its vertex updating function to compute new vertex values. The value propagator can perform graph traversals simultaneously for a list of source vertices.

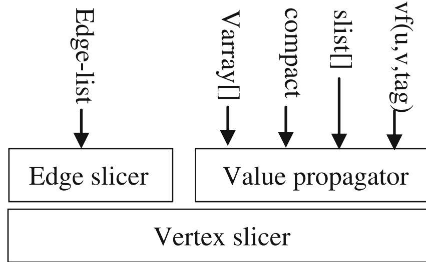


Fig. 1. Prototype of the topology-aware value propagating framework.

In a large-scale graph traversing application <V, E, S, f>, the framework is initialized by providing the input graph's edge list to the edge slicer. The application then can call the value propagator to perform graph traversals for a list of sources. It is required to provide four arguments to the value propagator: **slist**[], **compact**, **varray**[], and **vf**(*u*, *v*, *tag*). "**slist**[]" is the list of sources, and its length cannot be more then 64. "**varray**[]" is an array for storing the input graph's vertex values. If "**compact**" is true, **varray**[*i*] is the value of the *i*-th vertex; otherwise, "**varray**[]" saves 64 elements for every vertex, **varray**[*i**64 + *j*] is the *j*-th element of the *i*-th vertex. "**vf**(*u*, *v*, *tag*)" is the vertex updating function, which is called by the framework to update value on the *v*-th vertex when value of the *u*-th vertex is propagated to the *v*-th vertex. If "**compact**" is true, "*tag*" is always zero when "**vf**(*u*, *v*, *tag*)" is called; otherwise, it is a bitmap to indicate which elements are to be updated on the *v*-th vertex. If there are more than 64 sources in S, the application is required to divide its sources into groups. Each group contains at most 64 sources, and requires one call to the value propagator.

In the prototype, the input graph is represented as a vertex list and an adjacent list. In the vertex list, every element represents one vertex, and includes a type flag, a triplet (*offset*, *ideg*, *odeg*), and a visiting status vector. The type flag is to distinguish between dots, terminals, hub vertices, vertices on the path between a terminal and the hub vertex, and the netlike graph's vertices except the hub vertices. The vertex saves identifiers of its neighbors in the adjacent list; and the triplet (*offset*, *ideg*, *odeg*) describes addresses of these identifiers in the adjacent list, where *offset* is the first identifier's address, *ideg* is number of its ingoing neighbors and *odeg* is number of its outgoing neighbors. If the input graph is undirected, then *ideg* and *odeg* are equal. The visiting status vector consists of 64 bits, where the *i*-th bit denotes whether the vertex has been visited by the source specified by "**slist**[*i*]".

When the framework is initialized with the input graph's edge list, the edge slicer and vertex slicer cooperate automatically to initialize the vertex list and the adjacent list. When the value propagator is called, it first resets the visiting status vector of every vertex, then automatically propagates values on the input graph and calls the vertex updating function to update values in "varray[]".

5 Experimental Evaluation

This section presents the experimental results for various real-world and synthetic graphs. The platform for the experiments is a Dell R820 server. The server has 4 Intel(R) Xeon(R) E5-4640 CPUs and 256 GB memory. Each CPU has 8 physical cores sharing 20 M LLC, and can support 16 parallel threads with hyper-thread. In the experiments, the server is configured as a NUMA with 4 nodes.

We used four real graphs and one synthetic graph, as shown in Table 1. The synthetic graph is denoted as Kro_26_16. It was generated with the Kronecker model implemented in Graph500. When the graph was created, the scale parameter was set to be 26 and the edge-factor was 16. All the five graphs are assumed to be undirected. For each graph, we have also counted the vertices and edges of its netlike graph, denoted as NG's vertices and NG's edges respectively.

Table 1. Graphs used in the experiments.

	Vertices	Edges	NG's vertices	NG's edges
wikipedia	27,154,800	601,038,301	4,751,326	4,680,898
com-friendster	65,608,368	1,806,067,136	13,867,424	13,867,424
socfb-konect	59,216,216	92,522,017	20,959,355	54,348,978
twitter_rv	61,578,416	1,468,365,182	39,724,449	1,465,994,803
Kro_26_16	67,108,862	1,073,741,824	42,663,341	8,345,041

We developed two typical applications to evaluate the prototype and these three strategies presented in Sect. 3. One application is to estimate diameter of the input graph, requiring that every vertex saves one value to indicate its longest distance to other vertices. Another is to construct the BFS trees for a set of source vertices, requiring that every vertex saves one value for each source vertex. For each application, we have tested performance of five versions independently. One version is provided by the Ligra [15]. We denote it as the Ligra version. This version is implemented with OpenMP. It divides the input graph's vertices into equal chunks, and schedules these chunks dynamically to balance workload between processors. Other four versions were self-developed.

- **OCL version**. This version uses the owner-computing rule to partition the computation. Every computing node equally divides its local vertices into 16 chunks, and every chunk is statically assigned to one processor. When the application is executed, every processor propagates values from current frontiers to its chunk with the pulling mechanism, and updates these vertices accordingly.

- **EP version**. This version enhances the OCL version with the topology-based edge partitioning strategy. The input graph is partitioned into a netlike graph and a set of linear segments. Accordingly, computation on the input graph is divided into three phases. The first phase is to execute type-I tasks, and the last phase to execute type-III tasks. In the second phase, computation on the netlike graph is executed just as the OCL version does.
- **VP version**. This version enhances the EP version with the locality-based vertex partitioning strategy. It decomposes computation on the input graph into type-II tasks, and schedules these tasks dynamically. However, tasks in **tque**(*cpn*) can be assigned only to local processors of the computing node *cpn*.
- **ST version**. This version enhances the VP version with the work stealing technique. After all tasks in **tque**(*cpn*) have been assigned, it allows processors on the computing node *cpn* to steal tasks from other computing node.

On each graph, we randomly selected 192 vertices as the source vertices. Every implementation was repeated 10 times to traverse from these 192 source vertices on the input graph. The average time cost is the experimental result of the implementation for the input graph. We failed to run the Ligra versions of these two applications on soc-friendster, because this graph is too large for Ligra.

5.1 Experimental Results for Estimating Graph Diameter

In this experiment, every implementation equally divides the 192 source vertices into 3 groups, the traversing trees for all 64 sources of the same group are constructed simultaneously. Table 2 is the time costs of different versions, where the time unit is second.

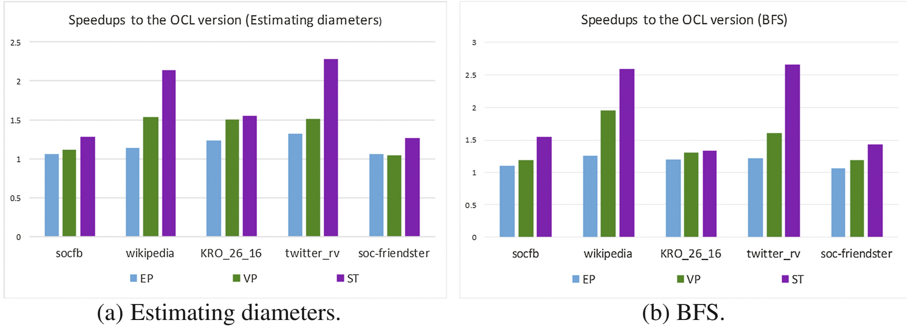
Table 2. Time costs for estimating diameters of different graphs.

	Socfb	Web-wiki	KRO_26_16	Twitter_rv	Soc-friendster
Ligra	9.63	44.79	55.8	140.79	–
OCL	5.64	17.88	16.11	42.3	88.17
EP	5.34	15.63	12.93	31.8	82.65
VP	5.01	11.64	10.68	27.81	84.45
ST	4.38	8.34	10.38	18.57	69.45

Although the Ligra version exploits the dynamic scheduling technique to balance workload, the OCL version has achieved significant better performance for all the first four graphs. On twitter_rv, speedup of the OCL version to the Ligra version is up to 3.46. The performance improvement tends to increase as the input graph's size increases. This results show that the performance bottleneck for most graph traversals is data accessing efficiency instead of load balance.

We evaluate the strategies presented in Sect. 3 with speedups to the OCL version. Figure 2(a) illustrates the results. Our topology-aware framework has achieved an overall speedup from 1.28 to 2.28. The edge partitioning strategy and work stealing strategy are effective for all the five graphs. The locality-based vertex partitioning strategy is ineffective for the synthetic graph. This is because that the OCL version

partitions computation almost equally between processors of the same computing node. We believe this case only occurs by chance. Result of the ST version shows that the owner-computing rule cannot balance workload between computing nodes.



(a) Estimating diameters. (b) BFS.

Fig. 2. Speedups to the OCL version.

5.2 Experimental Results for BFS

In this experiment, the Ligra BFS can process only one source vertex every time. To process 192 sources for every input graph, we have inserted a loop in the source code. Each of the four self-developed versions equally divides the 192 source vertices into 3 groups, the traversing trees for all 64 sources of the same group are constructed simultaneously. Table 3 is the time costs of different versions, where the time unit is second.

Table 3. Time costs for BFS of different graphs.

	Socfb	Web-wiki	KRO_26_16	Twitter_rv	Soc-friendster
Ligra	75.6	126.99	96	307.71	–
OCL	13.8	53.1	29.76	118.29	225.21
EP	12.51	42	24.87	97.41	211.89
VP	11.64	27.21	22.74	73.62	189.15
ST	8.91	20.52	22.47	44.34	156.78

The OCL version's performance is significantly better than that of the Ligra version, due to two factors. One is that Ligra version's locality is too poor, as shown in the previous subsection. Another is that the OCL version constructs 64 BFS trees simultaneously, resulting that value propagating overhead are shared between different sources.

We evaluate the strategies presented in Sect. 3 with speedups to the OCL version. Figure 2(b) illustrates the results. Our topology-aware framework has achieved an overall speedup from 1.32 to 2.67. All the three strategies are effective for all the five graphs.

6 Related Work

In recent years, many graph processing frameworks have been developed, including Pregel [11], Pregel+ [19], Giraph [5], Giraph++ [17], GraphLab [9], GraphX [18], and PowerGraph [7]. Although these systems are general enough to support different kinds of graph algorithms, there is no single system that has superior performance in all cases, two phenomena widely exists [10, 12]. One is that a general framework often shows different performance for different graph algorithms. Another is that an application's performance also often varies on different graphs. These phenomena are among the motivations behind this paper's work.

Data accessing efficiency and load balance are two key factors that hinder performance of graph applications, and are widely studied. Pregel introduced the message combining mechanism to reduce data-exchanging overhead between machines. Pre-gel+ introduced the vertex mirroring mechanism to reduce accesses to remote data. These two works give us beneficial hints for sharing value propagating overhead be-tween different sources. The difference is that our work focuses on data exchanging between vertices instead of machines. In GPS [13], the authors developed the large adjacency-list partitioning schema and dynamic repartitioning scheme to improve load balance. However, these two schemas often sacrifice locality, and in turn de-creases the performance, as shown by the Ligra's results in this paper. Giraph uses the multithreading method to maximize resource utilization. Experiments results in [20] disclose that sequential remote accesses can be faster than random local accesses. Our work exploits the vertex partitioning strategy to reduce random accesses, and improves load balance with the double-queue task scheduling strategy.

Different graph partitioning techniques have also been proposed to partition graph computation. PowerGraph has proposed the vertex-cut partitioning technique to improve load balance between tasks. GraphLego [21] replaces the traditional vertex-centric or edge-centric graph partitioning with a 3D cube model, so as to partition graph at the granularity of subgraphs. In [8], a graph transformation is proposed to reduce a large input graph into a small graph, so as to decompose computation on the original input graph. Our work combines the edge partitioning strategy and vertex partitioning strategy to partition graph computation.

7 Conclusion

Graph traversing is a value propagating pattern used by representative graph applications. This paper presents an efficient value propagating framework for large-scale graph traversing applications. It enables a graph computation to be partitioned according to both topology of the input graph and memory distribution of the graph's vertices. In this work, we propose to partition graph computation by combining edge partitioning and vertex partitioning. The proposed edge partitioning strategy is beneficial to reduce value propagating overhead. The proposed vertex partitioning strategy is beneficial to improve locality on each processor. To balance workload between processors and improve data

accessing efficiency, a greedy task scheduling strategy was devised. We have developed a prototype for the topology-aware graph traversing framework. The prototype was evaluated with two typical graph applications and five graphs. The experimental results show that this prototype has obvious better performance than Ligra. We also have evaluated the effectiveness of the strategies presented in this paper. Comparing with the owner-computing rule, the framework presented in this paper has an overall speedup from 1.28 to 2.67.

Acknowledgements. This work was supported by the National Key Research and Development Program of China (2016YFB0201900), and the National High Technology Research and Development Program ("863" Program) of China (Grant No. 2015AA015305).

References

1. Aingworth, D., Chekuri, C., Motwani, R.: Fast estimation of diameter and shortest paths (without matrix multiplication). SIAM J. Comput. **28**(4), 1167–1181 (1996)
2. Borassi, M., Crescenzi, P., Habib, M., Kosters, W.A., Marino, A., Takes, F.W.: Fast diameter and radius bfs-based computation in (weakly connected) real-world graphs. Theor. Comput. Sci. **586**(C), 59–80 (2015)
3. Brin, S., Page, L.: The anatomy of a large-scale hypertextual Web search engine. Comput. Netw. ISDN Syst. **30**(1–7), 107–117 (1998)
4. Chechik, S., Larkin, D.H., Roditty, L., Schoenebeck, G., Tarjan, R.E., Williams, V.V.: Better approximation algorithms for the graph diameter. In: Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms, pp. 1041–1052. Society for Industrial and Applied Mathematics, Philadelphia (2014)
5. Ching, A., Edunov, S., Kabiljo, M., Logothetis, D., Muthukrishnan, S.: One trillion edges: Graph processing at facebook-scale. Proc. VLDB Endow. **8**(12), 1804–1815 (2015)
6. Crescenzi, P., Grossi, R., Lanzi, L., Marino, A.: On computing the diameter of real-world directed (weighted) graphs. In: Klasing, R. (ed.) SEA 2012. LNCS, vol. 7276, pp. 99–110. Springer, Heidelberg (2012). doi:10.1007/978-3-642-30850-5_10
7. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: PowerGraph: distributed graph-parallel computation on natural graphs. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, pp. 17–30. USENIX Association, Berkeley (2012)
8. Kusum, A., Vora, K., Gupta, R., Neamtiu, I.: Efficient processing of large graphs via input reduction. In: Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, pp. 245–257. ACM, New York (2016)
9. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed graphlab: A framework for machine learning and data mining in the cloud. Proc. VLDB Endow. **5**(8), 716–727 (2012)
10. Lu, Y., Cheng, J., Yan, D., Wu, H.: Large-scale distributed graph computing systems: an experimental evaluation. Proc. VLDB Endow. **8**(3), 281–292 (2014)
11. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, pp. 135–146. ACM, New York (2010)

12. Nai, L., Xia, Y., Tanase, I.G., Kim, H., Lin, C.Y.: GraphBIG: understanding graph computing in the context of industrial solutions. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, p. 69. ACM, New York (2015)
13. Salihoglu, S., Widom, J.: GPS: A graph processing system. In: Proceedings of the 25th International Conference on Scientific and Statistical Database Management, p. 22. ACM, New York (2013)
14. Shun, J.: An evaluation of parallel eccentricity estimation algorithms on undirected real-world graphs. In: Proceedings of the 21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1095–1104. ACM, New York (2015)
15. Shun, J., Blelloch, G.E.: Ligra: A lightweight graph processing framework for shared memory. ACM Sigplan Not. **48**(8), 135–146 (2013)
16. Skeina, B.S.: The Algorithm Design Manual, 2nd edn. Springer, Heidelbergz (2008)
17. Tian, Y., Balmin, A., Corsten, S.A., Tatikonda, S., Mcpherson, J.: From think like a vertex to think like a graph. Proc. VLDB Endow. **7**(3), 193–204 (2013)
18. Xin, R.S., Gonzalez, J.E., Franklin, M.J., Stoica, I.: GraphX: a resilient distributed graph system on Spark. In: First International Workshop on Graph Data Management Experiences and Systems, p. 2. ACM, New York (2013)
19. Yan, D., Cheng, J., Lu, Y., Ng, W.: Effective techniques for message reduction and load balancing in distributed graph computation. In: Proceedings of the 24th International Conference on World Wide Web, pp. 1307–1317. ACM, New York (2015)
20. Zhang, K., Chen, R., Chen, H.: NUMA-aware graph-structured analytics. In: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 183–193. ACM, New York (2015)
21. Zhou, Y., Liu, L., Lee, K., Pu, C., Zhang, Q.: Fast iterative graph computation with resource aware graph parallel abstractions. In: Proceedings of the 24th ACM International Symposium on High-Performance Parallel and Distributed Computing, pp. 179–190. ACM, New York (2015)