Kimbap: A Node-Property Map System for Distributed Graph Analytics

Hochan Lee
hochan@utexas.edu
The University of Texas at Austin
Austin, Texas, USA

Roshan Dathathri
roshan.dathathri@microsoft.com
Microsoft Research
Redmond, Washington, USA

Keshav Pingali
pingali@cs.utexas.edu
The University of Texas at Austin
Austin, Texas, USA

Abstract

Most distributed graph analytics systems such as Gemini, Gluon, and SympleGraph support a computational model in which node properties are updated iteratively using properties of *adjacent* neighbors of those nodes. However, there are many algorithms that cannot be expressed in this model, such as the Louvain algorithm for community detection and the Shiloach-Vishkin algorithm for connected components. These algorithms may be more efficient or may produce better quality output than simpler algorithms that can be expressed using updates only from adjacent vertices.

This paper describes Kimbap, a distributed graph analytics programming framework, and its high-performance implementation that addresses this problem. Kimbap supports *general* vertex-centric algorithms by permitting the computation at a node to read and write properties of *any* node in the graph, not just its adjacent neighbors. The programming model allows programmers to specify iterative graph analytics applications, while the Kimbap compiler automatically generates the required communication code, and the Kimbap runtime organizes and synchronizes node-property pairs across the distributed-memory machines. The underlying system uses a distributed node-property map that is optimized for highly concurrent sparse reductions by using a graph-partition-aware sparse representation and by avoiding thread conflicts, thereby eliminating a major bottleneck that throttles performance in systems like Pregel that also support general vertex programs. Our experiments on CPU clusters with up to 256 machines (roughly 12000 threads total) show that (1) Louvain clustering algorithm in Kimbap is on average 4× faster than the state-of-the-art hand-optimized implementation for the same algorithm and (2) Kimbap matches or outperforms the state-of-the-art distributed graph analytics system for algorithms that can be expressed in both systems.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0385-0/24/04
https://doi.org/10.1145/3620665.3640421

ACM Reference Format:
Hochan Lee, Roshan Dathathri, and Keshav Pingali. 2024. Kimbap: A Node-Property Map System for Distributed Graph Analytics. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3620665.3640421

1 Introduction

Graph analytics systems today need to process graphs with billions of nodes and trillions of edges. Graphs of this size do not fit in the main memory of a single machine, so systems like Pregel [58], GraphLab [56], PowerGraph [41], Gemini [92], Gluon [27], and SympleGraph [93] have been developed to perform graph analytics on large clusters. These systems partition the graph so that each partition fits in the memory of a single host and support a vertex-centric programming framework that hides the details of partitioning and synchronization. In Pregel, a node in the graph can send or receive messages from any other node (we use *node* and *vertex* interchangeably). Several subsequent systems [27, 41, 56, 92, 93] restricted the dissemination of information to *a*djacent nodes. In these systems, nodes have properties and the property value at a node is updated by applying an *operator* that reads and/or writes properties of adjacent neighbors of that node [67, 70]. We call such operators *adjacent-vertex operators* and algorithms that can be expressed using only such operators *adjacent-vertex programs* [41]. By exploiting this restriction, these systems cache or *mirror* adjacent neighbors in a partition to optimize computation and communication, while supporting more advanced partitioning policies like vertex-cuts [27, 41]. Thus, these systems trade-off expressiveness for implementation efficiency.

Adjacent-vertex operators are not sufficient to express many graph algorithms for problems such as connected components [76], community detection [13, 79], spanning forests [15], clustering [55, 88], vertex covers [9, 31], similarity [49], etc. These algorithms access the properties of a node whose ID is computed dynamically; e.g., by using another node's property or by walking multiple hops in the topology. The operator in these algorithms requires accessing properties of *any* node in the graph. In this paper, we call such operators *trans-vertex operators*. Reductions (like

sum or minimum) on the node properties are common in trans-vertex operators.

Algorithms that use trans-vertex operators may be more efficient or may produce better quality output than adjacent-vertex programs for the same graph problem. For connected components for example, algorithms that use trans-vertex operators like Shiloach-Vishkin [76] are faster [53, 64] for high-diameter graphs than adjacent-vertex programs. Similarly, for minimum spanning forest, Boruvka's algorithm [15] uses trans-vertex operators and variants of it are faster [44, 59] on parallel hardware than algorithms limited to adjacent-vertex operators like Prim's algorithm [68]. On the other hand, for community detection, different algorithms may produce different outputs, and the quality of the output is a key determining factor in choosing the algorithm. The Louvain algorithm [13] uses trans-vertex operators, whereas the label propagation algorithm for community detection [72] uses only adjacent-vertex operators. Many studies [34, 52, 87, 91] have shown that Louvain produces better quality communities than the label propagation algorithm, so it is used in applications such as discovering structures in biological networks [60], financial networks [71], social networks [45], and citation networks [89].

NetworkX [43] has functions for many trans-vertex operators but is limited to a single machine. Distributed-memory implementations exist for some trans-vertex operators [12, 38, 78], but they are hand-optimized for specific algorithms and do not provide general systems support for trans-vertex operators. Graph databases like Neo4j [3] and TigerGraph [6] include some trans-vertex programs in their library but their languages like Cypher [36], GSQL [30], and GQL [66] do not support trans-vertex programs.

Trans-vertex operators can be implemented using general-purpose distributed key-value store systems [4, 35] or distributed shared memory systems [22, 62]. These systems do not natively support reductions but support atomic compare-and-swap (CAS) operations that can be used to implement reductions. For power-law graphs that have a small number of very high degree vertices, a large number of threads might be reducing property values for the same key or vertex. When CAS operations are used for such highly concurrent reductions, they may lead to conflicts. Consequently, these systems are not efficient for global, sparse, highly concurrent reductions. While Pregel supports trans-vertex operators [86], its implementation also suffers from conflicts during such reductions for power-law graphs [41].

The main challenge in building systems for trans-vertex operators is to support highly concurrent reads and reductions of properties of dynamically computed nodes, while exploiting the locality inherent in adjacent-vertex operators. In this paper, we present Kimbap, a general vertex-centric programming framework that supports general partitioning policies and is optimized for both adjacent-vertex and trans-vertex operators. The Kimbap system provides a novel distributed, concurrent node-property map that exploits locality for efficient reads, writes, and conflict-free reductions of arbitrary node properties. Kimbap supports the bulk-synchronous parallel (BSP) programming model, which allows node properties to be requested, read, or reduced efficiently in a "bulk" fashion. The property accesses can be sparse and at the end of each BSP step, reductions to node properties are performed collectively.

Kimbap provides a shared-memory vertex-centric abstraction for modifying node-property maps. Kimbap's compiler splits the computation operator specified by the programmer, generates BSP code along with the required request and reduce synchronization, and optimizes the generated code by detecting and removing unnecessary requests and their synchronization. The compiler also detects adjacent-vertex operators and specializes the generated code for them.

We implement seven graph algorithms in total for four graph problems using Kimbap. Five of them cannot be expressed using only adjacent-vertex operators. Ours is the first distributed implementation for the Leiden algorithm. Kimbap is evaluated using four graphs including the largest publicly available graph [61]. We demonstrate that Kimbap scales well on CPU clusters with up to 256 hosts. Kimbap is on average 4× faster than the state-of-art hand-optimized implementation [38] of the Louvain algorithm. This shows that Kimbap's abstraction does not come at the cost of performance. We also compare the Kimbap and Gluon [27] implementations of an adjacent-vertex program for connected components. Kimbap's performance is comparable, showing that Kimbap is efficient even for the simpler case of adjacent-vertex programs.

In summary, the paper makes the following contributions:

- We propose Kimbap, which is a programming framework for writing general vertex-centric programs at a high level of abstraction and executing them efficiently on distributed-memory machines (Section 3).
- We present a runtime and distributed execution model that efficiently manages reductions to node-property maps and their synchronization (Section 4).
- We implement a novel distributed, concurrent node-property map: (1) a novel representation for locality-optimized reads that leverages graph partitioning information and (2) a novel implementation for conflict-free reductions that allows reading the reduced values in the next BSP round (Section 4).
- We introduce a compiler that automatically generates the required, optimized communication and synchronization code from shared-memory Kimbap programs (Section 5).
- We evaluate our system on CPU clusters with up to 256 hosts and show that it significantly outperforms hand-optimized graph analytics implementations (Section 6).

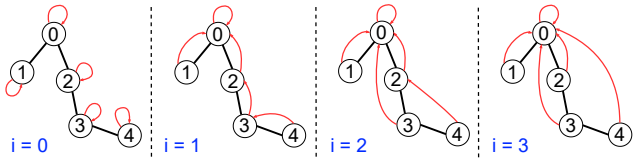


Figure 1. Illustration of executing Shiloach-Vishkin [76] connected components (CC-SV) on an example graph.

2 Background

This section presents background on distributed graph analytics by briefly describing vertex-centric programming frameworks and their implementations. This section also provides a brief description of the control-flow analysis that is used in our system.

2.1 Vertex-Centric Programming Frameworks

Pregel [58] introduced the vertex-centric programming framework for distributed graph analytics. Execution occurs in rounds and in each round, a node can process messages received in the previous round and send messages to any other node. Nodes have properties (a state in custom data types) but a node can access only its own properties. Subsequent frameworks like GraphLab [56] and Gluon [27] raised the level of abstraction from message passing to shared-memory while also restricting the dissemination of information to *adjacent* nodes. In these frameworks, node properties are updated by operators that iterate over *active* nodes, and read and write the properties of a node and its *adjacent* neighbors. Some frameworks like PowerGraph [41] and Gemini [92] retain the notion of message passing but restrict them to *adjacent* neighbors. We call both these variants, *adjacent-vertex frameworks*.

Adjacent-vertex frameworks have been used to design algorithms for many graph analytics problems but it has its limitations. Consider *pointer jumping*, which is also known as the *shortcut* operation. Each node n_1 has a property consisting of the ID of some other node n_2 in the graph, which is said to be the *parent* of n_1. The shortcut operation sets $parent(n_1)$ to $parent(parent(n_1))$, i.e., $parent(n_2)$. As n_2 can be any node in the graph, the shortcut operator for n_1 may need to read the parent of a non-adjacent node, which cannot be expressed in adjacent-vertex frameworks. Figure 1 shows an example. Black edges in this figure correspond to edges in the input graph and red edges show the *parent* property of each node. In the third graph from the left ($i = 2$), $parent(4) = 2$ and $parent(2) = 0$. Applying the shortcut operation to node 4 sets $parent(4)$ to 0, as shown in the fourth graph ($i = 3$).

Pointer-jumping is frequently used as a basic operation in efficient algorithms for problems such as minimum spanning forest [15, 26] and connected components [76, 78]. Here we describe the Shiloach-Vishkin algorithm [76] for connected components, which we use as the running example in this paper. Each node n has a *parent* property which is set to n at the start of the algorithm, as shown in the first graph from the left ($i = 0$) in Figure 1. At the end of the algorithm, $parent(n)$ is the smallest ID of *any* node in the same component as n. In Figure 1, there is a single component and the smallest ID is 0, so at the end of the algorithm, the *parent* property of all nodes is 0, as shown in the fourth graph ($i = 3$).

The Shiloach-Vishkin algorithm works in rounds. In each round, the *shortcut* operation is applied to each node after the application of the *hook* operation to each node. To apply the *hook* operator to a node n, all directed edges $n \rightarrow m$ of a symmetric graph are examined. Let $p = parent(n)$; if $p > parent(m)$, then $parent(p)$ is *min*-reduced by $parent(m)$; i.e., the minimum of $parent(p)$ and $parent(m)$ is written to $parent(p)$. The algorithm terminates when no *parent* value is updated in a round. The hook operator for n may need to read and write the parent value of p, which may be any node in the graph, not necessarily a neighbor of n. This algorithm clearly cannot be expressed in adjacent-vertex frameworks.

2.2 Vertex-Centric Distributed Graph Systems

To implement vertex-centric frameworks on distributed clusters, the graph is partitioned between the hosts of the cluster [73]. Each host performs computations on nodes in its own partition but must communicate with other hosts to ensure that the properties of nodes on the boundaries of partitions are synchronized. To highlight the implementation differences that arise due to expressiveness, we will use Gluon [27] and Pregel [58] as examples of adjacent-vertex frameworks and general vertex-centric frameworks respectively.

In Gluon, edges are partitioned among hosts and *proxy nodes* are created for their end points. Gluon provides several partitioning policies. These are classified into *edge-cut* policies, which assign all incoming edges or all outgoing edges of a node to the same partition, and *vertex-cut* policies, in which edges of a node can be on different partitions.

Since the edges connected to a given node may be partitioned between different hosts, there may be several proxy nodes for a given node in the graph. One of these proxies is selected as the *master* node and the others are called *mirror* nodes. Intuitively, the master node holds the canonical property value for that node, and it is responsible for updating the values at mirror nodes if its property value changes. In this way, each partition of the graph is a small graph in itself, and the operator can be executed without worrying about the fact that the graph is partitioned across machines, thereby exploiting locality and decoupling computation and communication.

To ensure that mirror node properties are updated efficiently, Gluon uses the bulk-synchronous parallel (BSP) computing model. The algorithm is executed in rounds, and each round consists of a computation phase followed by a

communication phase. In the communication phase, property values at mirror nodes are reduced at the master node, and the resulting value is broadcast to the mirror nodes. This general mechanism works for all partitioning policies. Gluon includes an optimization to elide reduce or broadcast depending on the graph algorithm by exploiting the structural invariant in the partitioning policy. An example is the following: if the partitioning policy is an outgoing edge-cut (OEC), then mirror nodes have no outgoing edges; a push-style graph algorithm only reads the property values for nodes that have outgoing edges, so property values of mirror nodes are not read; Gluon therefore reinitializes a mirror node's value at the end of the BSP step to the identity value of the reduction, instead of updating it to the value at the master node, which would require communication. There are similar optimizations for other structural invariants in partitioning policies. Gluon also exploits the temporal invariant in the partitioning policy (that the partitioning does not change during algorithm execution) to only send the node property values that have been updated while minimizing the metadata for those nodes. Gluon's partitioning invariant optimizations reduce both the number of communication messages and the communication volume, thereby reducing communication overheads.

In Pregel, nodes are partitioned among hosts and each node can directly access the edges adjacent to it; i.e., it only supports *edge-cut* policies. Messages are buffered for delivery to a particular host and they are combined/reduced when they are added to the buffer. This could lead to severe contention for very high-degree vertices in power-law graphs [41]. Computation and communication are thus tightly coupled. Even if the property values have not been updated, messages between nodes must be resent for the property values to be accessed, which can significantly hurt performance for some adjacent-vertex operators. Pregel's message passing framework prevents it from caching mirror node properties and providing direct access to master or mirror node properties. Pregel also does not support *vertex-cut* policies that can help scaling out execution on power-law graphs [25, 40, 41, 46].

2.3 Control-Flow Analysis

We briefly describe some standard terms in control-flow analysis that are used in the exposition of the Kimbap compiler described in Section 5.

A control-flow graph is a representation of all paths that might be traversed in a program during its execution [8]. In this paper, we consider the *statement-level* control-flow graph in which each node represents a single statement, and a directed edge represents control-flow between statements. A node may have more than one immediate successor if there is conditional flow of control from that node to its immediate successors.

```
template <typename PropTy>
class NodePropMap {
public:
  NodePropMap(Graph& graph);

  PropTy Read(Node key);

  void Reduce(Node key, PropTy value,
              function<PropTy(PropTy, PropTy)>& op);

  void Set(Node key, PropTy value);

  ...
}
```

Figure 2. Node-property map API exposed to developers.

There are two special blocks in the control-flow graph: (1) the entry block through which all control flow enters, and (2) the exit block through which all control flow leaves. A node M *dominates* a node N if every path from the entry block to N passes through M (the entry block does not have a dominator by definition). Dominance is a tree-structured relation, and the tree is called the *dominator tree*. The root of this tree is the entry node. The parent of a node N in this tree is called the immediate dominator of N (this relation is not defined for the entry node). Similarly, a node N is a *post-dominator* of a node M if every path from M to the exit block passes through N. Post-dominance is also a tree-structured relation, and the *immediate post-dominance* relation can be defined analogously.

3 Kimbap Programming Framework

Section 3.1 describes the API for the node-property map. Section 3.2 describes a parallel construct we introduce for writing vertex-centric programs in Kimbap. Section 3.3 illustrates this programming framework using the Shiloach-Vishkin algorithm [76] for connected components (CC-SV).

3.1 Node-Property Map API

In Kimbap, programmers use a concurrent map, called *node-property map*, to store node IDs and properties as key-value pairs. A node property for a master node is called *master node property*; for any other node, it is called *remote node property*. A *mirror node property* is a special case of a remote node property. A node-property map maintains canonical values of master node properties, and caches remote node properties. The API provided to programmers has a shared-memory view and hides the complexity of the distributed implementation by exposing only the functions shown in Figure 2.

- The constructor takes the graph as an input to enable each host to determine masters and mirrors on that host.
- `Read()` returns the property value of a given node.

```
KimbapWhile (<NodePropMap>) Updated
  ParFor (<Node-Iterator>) {
    <Operator>
  }
```

Figure 3. Parallel construct for vertex-centric programs.

- `Reduce()` reduces a given value onto the property value of a given node; the third parameter specifies the reduction function that takes two values as inputs and returns a combined value. This function must be associative and commutative.
- `Set()` assigns a specified value to a node. It is meant to be used only during initialization of the map, which does not have data races. If multiple values are written to the same node's property, the runtime is free to pick any one of them.

3.2 Parallel Construct for Vertex-Centric Programs

Figure 3 shows the parallel construct for specifying iterative vertex-centric shared-memory programs. The Kimbap compiler performs analysis and generates optimized code to run on distributed clusters. The construct has three programmer-defined components:

- *Quiescence condition*: this is specified by providing a node-property map. The parallel-for loop is executed repeatedly until there are no updates to that map. Extending the quiescence condition to other constructs is straightforward but is not needed for the applications of interest in this paper.
- *Node iterator*: in most applications, this is an iteration over all nodes in the graph but it is also possible to specify iteration over a subset of nodes.
- *Operator*: the operator can read and write properties of any node. To enable generating efficient code, we require the operator to be a *cautious operator* [67], which means that writes to property values must occur after all reads in the operator (similar to eliminating hold and wait for deadlock prevention). The only edges that can be accessed from the graph object are the edges connected to the active node. If programmers need to access other edges, they can store the edges themselves as a property on the nodes in a node-property map and use that map to access edges of any node.

While the writes to the node-property map are concurrent, the semantics of reads require only eventual consistency; i.e., the operator is resilient to stale reads because the operator will be applied until there are no updates.

Prior distributed vertex-centric graph systems also require cautious operators [27, 39, 41, 58] and require only eventual consistency [27–29, 81]. For adjacent-vertex operators, Kimbap is as programmable as prior works. Its abstraction is similar to that of Abelian [39] but unlike Abelian, Kimbap supports trans-vertex operators. Pregel [58] is the only prior distributed graph system that is as expressive as Kimbap because it is the only one that supports trans-vertex operators. However, Kimbap's shared-memory abstraction enables optimizations (Section 4.2) that cannot be accomplished using Pregel's message passing abstraction.

```
1  type ParentTy = Node;
2  type ParentNPM = NodePropMap<ParentTy>;
3
4  void Hook(ParentNPM& parent_npm,
5      Graph& graph, BoolReducer& work_done) {
6    KimbapWhile (parent_npm) Updated
7      ParFor (Node src : graph.Nodes()) {
8        ParentTy src_parent = parent_npm.Read(src);
9        for (Edge edge : graph.Edges(src)) {
10         Node dst = edge.DestinationNode();
11         ParentTy dst_parent = parent_npm.Read(dst);
12         if (src_parent > dst_parent) {
13           work_done.Reduce(true, logical_or);
14           parent_npm.Reduce(
15               src_parent, dst_parent, min);
16         }
17       }
18     }
19 }
20
21 void Shortcut(ParentNPM& parent_npm, Graph& graph){
22   KimbapWhile (parent_npm) Updated
23     ParFor (Node node : graph.Nodes()) {
24       ParentTy parent = parent_npm.Read(node);
25       ParentTy grand_parent = parent_npm.Read(parent);
26       if (parent != grand_parent) {
27         parent_npm.Reduce(node, grand_parent, min);
28       }
29     }
30 }
31
32 void CC_SV() {
33   Graph graph = /* load graph */;
34   ParentNPM parent_npm(graph);
35   BoolReducer work_done;
36
37   // Initialization.
38   ParFor (Node node : graph.Nodes()) {
39     parent_npm.Set(node, node);
40   }
41
42   do {
43     work_done.Set(false);
44     Hook(parent_npm, graph, work_done);
45     Shortcut(parent_npm, graph);
46   } while (work_done.Read())
47 }
```

Figure 4. Shiloach-Vishkin algorithm [76] for connected components (CC-SV) in Kimbap's programming framework.

3.3 Shiloach-Vishkin Connected Components (CC-SV) in Kimbap

Figure 4 shows the CC-SV algorithm written in Kimbap. ParentNPM is the node-property map that keeps track of the parent of each node (Line 2). This property map is initialized in Lines 38 to 40 using the *set* operator of the property map.

```
template <typename PropTy>
class NodePropMap {
public:
  ... // user visible functions

  // functions used by a compiler; hidden from users
  void ResetUpdated();
  void IsUpdated();
  void Request(Node key);
  void RequestSync();
  void ReduceSync();
  // functions used by a compiler to optimize
  void BroadcastSync();
  void PinMirrors();
  void UnpinMirrors();
}
```

Figure 5. Node-property map API for the compiler.

The hook operator in CC-SV is specified in Lines 8 to 17. The edges connected to each node `src` are examined (Line 9), and if a neighbor's parent is smaller than the parent of node `src` (Line 12), the operator reduces `src`'s parent's parent with the neighbor's parent. To express this in Kimbap, `Reduce()` is used with a *min* function (Line 15). This operator is executed for all nodes (Line 7) until there are no updates to the parent of any node (Line 6). The shortcut vertex operator can be specified similarly (Lines 24 to 28). The node reads its parent and its parent's parent (Lines 24 and 25) and updates its parent to its parent's parent if they are different (Line 26), using `Reduce()` with the *min* operator (Line 27). This operator is executed for all nodes (Line 23) until there are no updates to the parent property of any node (Line 22). Both hook and shortcut may need to be repeated if any of the nodes' component was updated during hook. The user can track this using a `BoolReducer` that provides a (distributed) concurrent reducer for a boolean value (details are omitted for brevity).

4 Kimbap Runtime

Section 4.1 describes the execution model of Kimbap and introduces the low-level node-property map API. Section 4.2 describes the optimized implementation of the node-property map and extends the API to enable compiler optimizations.

4.1 Distributed Execution Model

An asynchronous execution model may hide communication overheads, but may generate a large number of messages, generate duplicate messages, and yield high materialization overheads. Kimbap instead batches and de-duplicates messages, and minimizes costs for materializing the node-property maps. Consequently, Kimbap adopts a bulk synchronous parallel (BSP) execution model.

To exploit locality, Kimbap not only decouples computation and communication but also decouples reads of the node-property map from reductions to it. By caching remote node properties read during compute (instead of materializing them on-the-fly whenever they are accessed), Kimbap minimizes the materialization costs. Kimbap also includes optimizations (Section 4.2) to further reduce the overhead of materialization for both read and written remote node properties. As a result, the overhead of communication is also reduced.

The Kimbap compiler, described in Section 5, implements each `KimbapWhile` loop as a BSP program, using the low-level API shown in Figure 5. The BSP rounds are iterations of `KimbapWhile` and they are synchronized using `IsUpdated()`. Each round of this BSP program consists of 4 kinds of phases: *request-compute*, *request-sync*, *reduce-compute*, and *reduce-sync*. The reduced values in a round can only be read in the next round. Kimbap uses non-blocking sends and receives within each phase to reduce communication overheads.

In *request-compute*, a parallel loop on each host generates requests using `Request()` for properties of remote nodes. We use a concurrent bitset and set the ith bit if node i is requested, which avoids duplicate requests. Then each host calls the collective `RequestSync()`. In *request-sync*, we aggregate requests from the bitset, and send a single request message to every other host. As requests are received from the other hosts, a parallel loop on each host serves them by reading the canonical property values from the corresponding master nodes. Each host receives a single request from any individual host, and produces a single corresponding response message. Each host then materializes the memory required only for all the requested remote nodes. As responses are received from the other hosts, each host copies or caches properties in the materialized memory and can now read them directly in the *reduce-compute* phase.

In *reduce-compute*, a parallel loop on each host reads cached node property values using `Read()` and reduces them onto new node property values using `Reduce()`, which computes partially reduced values that cannot be read in this phase. Then each host calls the collective `ReduceSync()`. In *reduce-sync*, we perform scatter-gather-reduce (SGR). Partial results for nodes are scattered by each host to their owner hosts (one message between every pair of hosts). Each host then gathers these partial results (without explicit ordering) and reduces them in a parallel loop onto their master node property values. This *SGR* is further optimized as detailed in Section 4.2. After this, cached remote node properties on each host become stale, so they are dropped and they must be requested again before being read.

Adjacent-vertex frameworks like Gluon [27] do not have *request-compute* and *request-sync* phases because remote accesses are restricted to mirror nodes and they always cache mirror node properties. They also read and reduce to the cached values directly (using atomics) during *reduce-compute*. On the other hand, general vertex-centric frameworks like Pregel [58] use message passing instead of Kimbap's shared-memory node-property maps. As a consequence, (1) application programmers need to write operators for requests and responses explicitly; (2) they cannot decouple compute and

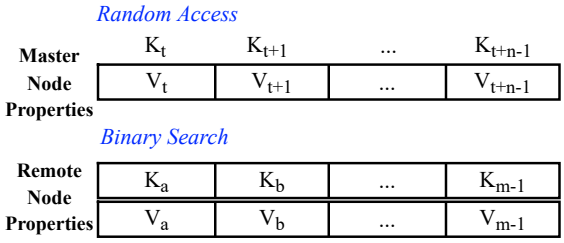


Figure 6. Graph-partition-aware representation (GAR) of node-property map on each host for read accesses.

sync for both request and reduce; and (3) they cannot provide direct access to cached remote node properties during compute. Pregel also does not de-duplicate requests.

4.2 Optimizations in Node-Property Map

Node-property maps can be implemented using off-the-shelf distributed in-memory key-value stores [4, 35]. Kimbap instead includes a custom implementation with novel domain-specific optimizations: (1) graph-partition-aware representation (GAR), (2) conflict-free reductions (CF), and (3) pinned mirrors (PM). These optimizations cannot be implemented in general key-value stores because they are specific to graph analytics. General vertex-centric frameworks like Pregel [58] can implement *CF* but the message passing abstraction in Pregel prevents it from implementing *GAR* and *PM*.

Graph-Partition-Aware Representation (GAR). We measured the node property reads in seven graph algorithms and two input graphs on 4 and 32 hosts (experimental setup details are presented in Section 6.1). On 4 hosts, 65% of the reads were on average for master node properties. On 32 hosts, 50% of the reads were on average for master node properties. This is very high since on 32 hosts, only about 3% of the nodes of the graph are master nodes on a given host. In addition, all of the master node properties were accessed at least once. Thus, there is significant locality in node properties accessed on a host. To leverage this locality, each host owns master node properties and caches remote node properties to read in Kimbap. As a consequence, a master node property is always materialized on a node-property map, but a remote node property should have been requested and is materialized during `RequestSync()` and dropped after `ReduceSync()`. We also use different in-memory layouts for master and remote node properties, as shown in Figure 6. For master nodes, we use a vector for their properties and do random access to read a value.

We use a custom map for remote nodes: a vector stores the nodes and another vector stores their properties. These vectors are sorted by the nodes and we do a binary search to read a property value.

Conflict-Free (CF) In-Memory Reductions. To avoid conflicts in distributed-memory, the node-property map uses

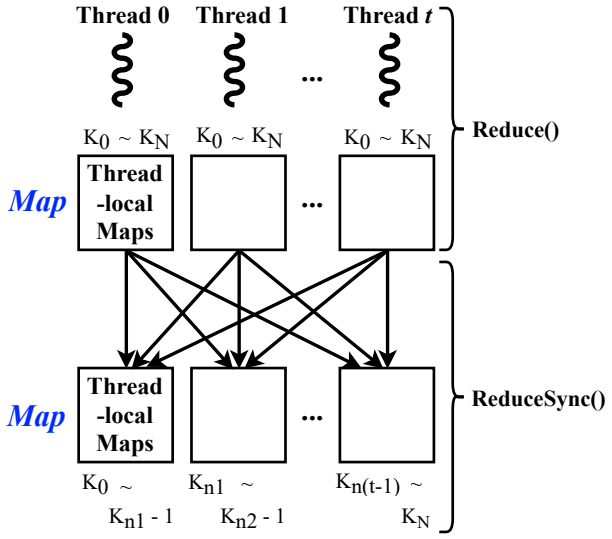


Figure 7. Node-property map on each host for concurrent write accesses: dataflow between its thread-local maps for conflict-free (CF) reductions ($K_i \sim K_j$ is the range of potential keys for that map).

scatter-gather-reduce (SGR) (Section 4.1). The implementation of the concurrent map on each host is further optimized to reduce access conflicts between threads. During the *reduce-compute* phase, each thread maintains its own thread-local map that is updated on `Reduce()` of both master and remote nodes, as shown in Figure 7. The same node or key can be present in multiple thread-local maps and each thread-local map can contain a value for any node in the graph. As the accesses are sparse, only a few nodes (that have high-degree) are expected to be in multiple thread-local maps. During the *reduce-sync* phase, these thread-local maps are combined in `ReduceSync()` before communication. This combining step is also optimized to reduce access conflicts. The runtime assigns a disjoint node range to each thread, and each thread traverses all the thread-local maps, combines partial results for its assigned nodes, and writes the final value to its new thread-local map. These new thread-local maps are thus constrained such that a node may be present in only one of the thread-local maps. These maps are then passed to the communication runtime for *scatter-gather-reduce*, which yields the updated values on the master node properties.

Pinned Mirrors (PM). The runtime also provides methods (Figure 5) that the compiler can use to further exploit locality. `BroadcastSync()` broadcasts each master node property value to all its mirror nodes. `PinMirrors()` materializes or caches the mirror node property values in the node property map and calls `BroadcastSync()`. `UnpinMirrors()` drops the mirror nodes from the node property map. These functions can be used to replace the two-way request-response for reading mirror node properties with one-way broadcast, which

reduces the communication volume and the number of communication messages. When pinned mirrors are enabled by the compiler, Kimbap incorporates Gluon's [27] partitioning invariant optimizations during broadcast to further decrease communication overheads. For adjacent-vertex operators, pinned mirrors are always enabled and they are essential for matching the performance of adjacent-vertex frameworks like Gluon. Pinned mirrors can also be enabled for some trans-vertex operators like the hook operator in Figure 8.

5 Kimbap Compiler

Kimbap's compiler relies on the node-property map API and the (implicit) graph API (such as `Nodes()` and `Edges()`) to analyze the operators. Figure 8 shows the optimized code generated by the Kimbap compiler from the CC-SV code in Figure 4. We use this as a running example to describe the required transformations and the conditional optimizations.

5.1 Transformations

The compiler builds a control-flow graph for the user-provided code and transforms it to: (i) introduce iterative loops with a termination condition, (ii) split the operator and introduce the required requests for the node properties, and (iii) insert the required request or reduce synchronization of node properties. Abelian [39], a compiler for adjacent-vertex programs, has similar transformations but it does not handle requests required for trans-vertex operators. These transformations are not applicable to Pregel [58] due to its abstraction.

DoWhile: The compiler translates Kimbap's parallel construct into a `ParFor` loop within a `do ... while`. `ParFor` is an OpenMP-style parallel-for loop. The termination condition for the `while` loop uses the `IsUpdated()` function of the given node-property map (Line 40 in Figure 8).

Split operator and request: The compiler analyzes each operator to find all `Read()` calls to any node property map. It iterates over these calls such that if R_1 dominates R_2, then R_1 will be iterated before R_2. For each call R, it finds the `ParFor` PF that dominates R but does not dominate any other `ParFor`. It then finds all operations O that dominate R and are dominated by PF. After the immediate dominator of PF, it inserts copies of PF, R, and all operations O replicating the dominance between them. It then replaces R's copy with the corresponding `Request()`. This ensures that (1) all accesses to the node property map are requested in a previous `ParFor` before being read and (2) if R_1 dominates R_2, R_1 will execute before the request corresponding to R_2 in the same `ParFor`. The second `ParFor` generated for the shortcut operator is shown in Lines 27 to 30. The first `ParFor` is optimized out, which is described in Section 5.2.

RequestSync and ReduceSync: After the request transformation, the compiler analyzes each operator (including new operators) to find all `Request()` or `Reduce()` calls. For each such call R, it finds the `ParFor` PF that dominates R and inserts a `RequestSync()` (Line 31) or `ReduceSync()` (Line 39) respectively before the immediate post-dominator of PF. This ensures that the requests and the reductions to the node-property map are synchronized.

```
1  void Hook(ParentNPM &parent_npm, Graph& graph,
2      BoolReducer& work_done) {
3    parent_npm.PinMirrors();
4    do {
5      parent_npm.ResetUpdated();
6      ParFor (Node src : graph.Nodes()) {
7        ParentTy src_parent = parent_npm.Read(src);
8        for (Edge edge : graph.Edges(src)) {
9          Node dst = edge.DestinationNode();
10         ParentTy dst_parent = parent_npm.Read(dst);
11         if (src_parent > dst_parent) {
12           work_done.Reduce(true, logical_or);
13           parent_npm.Reduce(
14               src_parent, dst_parent, min);
15         }
16       }
17     }
18     parent_npm.ReduceSync();
19     parent_npm.BroadcastSync();
20   } while (parent_npm.IsUpdated());
21   parent_npm.UnpinMirrors();
22 }
23
24 void Shortcut(ParentNPM &parent_npm, Graph &graph) {
25   do {
26     parent_npm.ResetUpdated();
27     ParFor (Node node : graph.MasterNodes()) {
28       ParentTy parent = parent_npm.Read(node);
29       parent_npm.Request(parent);
30     }
31     parent_npm.RequestSync();
32     ParFor (Node node : graph.MasterNodes()) {
33       ParentTy parent = parent_npm.Read(node);
34       ParentTy grand_parent = parent_npm.Read(parent);
35       if (parent != grand_parent) {
36         parent_npm.Reduce(node, grand_parent, min);
37       }
38     }
39     parent_npm.ReduceSync() ;
40   } while (parent_npm.IsUpdated());
41 }
```

Figure 8. Compiler-generated code for hook and shortcut functions of CC-SV in Figure 4.

5.2 Optimizations

The compiler uses novel domain-specific optimizations for two cases: (i) when edges are not accessed in the operator, like the *shortcut* operator of CC-SV, and (ii) when all reads in the operator are only to the active node and its adjacent neighbors, like the *hook* operator of CC-SV. These optimizations are not applicable to Abelian [39] as it does not handle requests. In Pregel [58], the first optimization can be implemented (by programmers) but the message passing abstraction prohibits the second optimization.

Master nodes RequestSync elision: Before the request transformation, the compiler analyzes the original operator to detect whether the operator accesses any edge of the

active node. Different partitions may contain different edges for the same node but if the edges are not accessed, then the computation only depends on properties of the nodes, which are consistent (synchronized) across partitions. Therefore, all partitions that contain the mirror node would compute the same updates as the master node, which is redundant computation and would lead to redundant communication. Hence, the compiler modifies the iterator to filter-out mirror nodes and restrict the iterator to master nodes (Line 27). After the request transformation, the compiler analyzes each operator and if the `Request()` call is only to master nodes, it deletes that operator and the corresponding **`ParFor`** and `RequestSync()`. The first **`ParFor`** for the *shortcut* operator is removed because of this optimization.

Adjacent neighbors RequestSync elision: Before the request transformation, the compiler analyzes the operator to detect whether all `Read()` accesses are to the node and its adjacent neighbors only. If so, it is more efficient to pin mirrors and broadcast every update rather than to request node properties. Therefore, the compiler inserts `PinMirrors()` before the immediate-dominating do-while loop (Line 3) and `UnpinMirrors()` after the loop (Line 21). After the ReduceSync transformation, the compiler inserts a `BroadcastSync()` (Line 19) after every `ReduceSync()`. Note that the writes/reduces could be to any node, not necessarily only to an adjacent neighbor of the active node. Due to this, requests are elided for the *hook* operator in CC-SV.

6 Evaluation

We evaluate Kimbap and compare it with the state-of-the-art implementations and frameworks[1]: distributed Louvain clustering in Vite [38], distributed connected components in Gluon [27], and a shared-memory graph analytics framework, Galois [64]. We also evaluate Kimbap programs implemented using an in-memory key-value store, Memcached [35].

6.1 Experimental Setup

We performed all experiments on the Stampede2 SKX cluster [77] at the Texas Advanced Computing Center (TACC) [7] which has a 2.1GHz Intel Xeon Platinum 8160 Skylake architecture, 28 cores each on two sockets (hyperthreading disabled), 192GB DDR4 RAM, 32KB L1 data cache per core, and 100Gbps Intel Omni-Path (OPA) network. The Kimbap system is implemented in C++ and was compiled with g++ 9.4.0. We used up to 256 CPU hosts with 48 threads per host.

[1]Pregel [58] is the only prior distributed graph processing system that supports trans-vertex operators, but it is not available publicly. Giraph [1], the open-source counterpart to Pregel, requires Hadoop, which is not supported on the Stampede2 cluster that we use for evaluation. Furthermore, the MIS and CC algorithms in the Giraph's library use only adjacent-vertex operators, and there are no implementations for Louvain, Leiden, or MSF in its library. Prior works [27, 41, 92] show that for adjacent-vertex programs, Gluon is faster than Gemini, Gemini is faster than PowerGraph, and PowerGraph is faster than Pregel. Pregel and Giraph also provide fault-tolerance and other features that are orthogonal to Kimbap's contributions and that may have high runtime overheads. Due to these reasons, we did not evaluate Pregel or Giraph.

	road-europe	friendster	clueweb12	wdc12
$\|V\|$	173M	41M	978M	3B
$\|E\|$	365M	2B	85B	256B
$\|E\|/\|V\|$	2	58	87	72
Max Degree	16	3M	7K	95B
Size (GB)	3	9	325	1K

Table 1. Input graphs and their statistics.

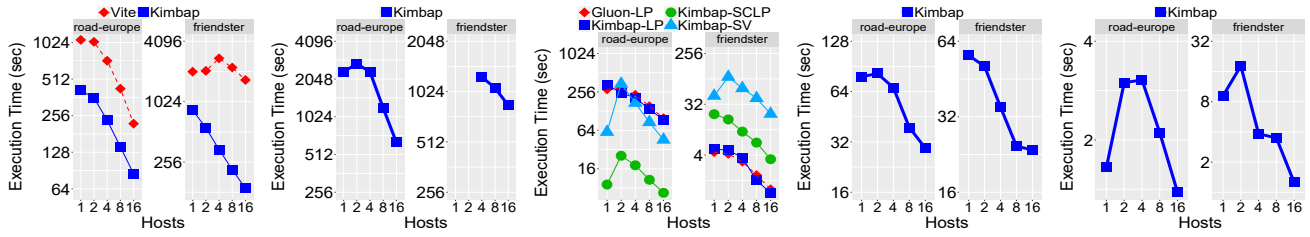
Application	Adjacent-Vertex Operator	Trans-Vertex Operator
LV	•	•
LD	•	•
MSF		•
CC-LP	•	
CC-SCLP	•	•
CC-SV		•
MIS	•	

Table 2. Operator types used in each application.

Table 1 lists the input graphs that we tested in our evaluations and their statistics. *road-europe* [37] is a road network graph, *friendster* [54] is a social network graph, and *clueweb12* [69] and *wdc12* [61] are web-crawl graphs. wdc12 is the largest publicly available graph. road-europe has a high diameter with roughly uniform and small node degrees. Others are power-law graphs that have a small number of very high-degree nodes. All graphs are symmetrized by adding reverse edges to represent undirected graphs. We categorized road-europe and friendster as medium size, and clueweb12 and wdc12 as large size. The medium size and the large size graphs were evaluated with up to 16 hosts and 256 hosts respectively.

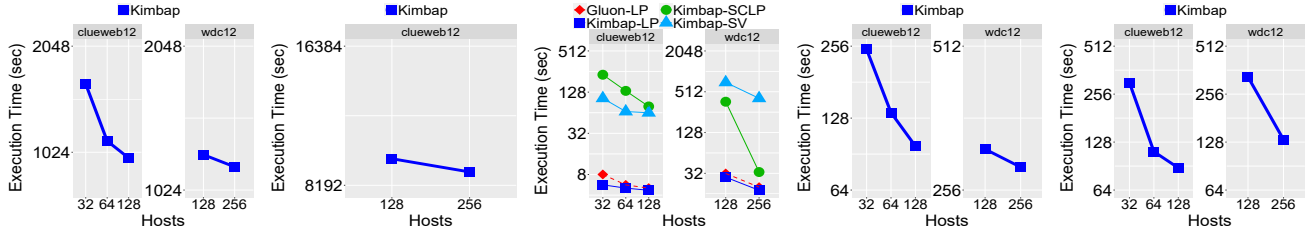
We consider 7 graph algorithms in total for 4 graph problems: community detection, connected components (CC), minimum spanning forest (MSF), and maximal independent sets (MIS). As Louvain (LV) and Leiden (LD) algorithms for community detection produce different outputs, we discuss their results separately. The adjacent-vertex and/or trans-vertex operators used in the algorithms are shown in Table 2. We report the execution time of each graph algorithm, excluding graph loading/partitioning time, as an average of 3 runs.

LV: We implemented the deterministic LV algorithm [13]. LV consists of two main phases: a clustering refinement and a graph coarsening based on clusters. In clustering refinement, each node calculates a score called the *modularity gain* [63] for each neighbor's cluster, by reading both the node's and the neighbor's clusters' properties, to determine whether it should move to that cluster. In Kimbap, a cluster's property is stored in its representative node's property. Both



(a) LV **(b)** LD **(c)** CC **(d)** MSF **(e)** MIS

Figure 9. Strong scaling (log-log scale) of Kimbap, Vite, and Gluon for medium size graphs (each host has 48 cores); missing points (LD) are due to out-of-memory (OOM).



(a) LV **(b)** LD **(c)** CC **(d)** MSF **(e)** MIS

Figure 10. Strong scaling (log-log scale) of Kimbap and Gluon for large size graphs; Vite timed-out after 9000 seconds.

Vite [38] and Kimbap use the same algorithm but Vite uses the early termination runtime optimization that excludes nodes from computation with 75% probability if they stay in the same cluster for 4 consecutive refinement phases. We did not implement this in Kimbap as our focus is on application-agnostic optimizations.

LD: We implemented the deterministic LD algorithm [79]. LD improves clustering quality from LV by splitting clusters into subclusters, calculating connectivity of the subclusters, and moving loosely connected subclusters to neighbor clusters. We use five node property maps for cluster and subcluster information. Our LD is the *first distributed implementation.*

CC: We implemented three CC algorithms: Shiloach-Vishkin (SV) [76], shortcutting label propagation (SCLP) [78], and label propagation (LP) [26]. Each node keeps track of its parent in a node-property map. We compare these with the state-of-the-art distributed LP implementation in Gluon [27].

MSF: We implemented Boruvka's [15, 26] MSF algorithm. The Boruvka algorithm finds minimum spanning trees (MSTs) through successive union-find and shortcut operations. We use two node-property maps in Kimbap. The first one keeps track of a parent node for each node and the second one stores a MST (value) that will be merged with the another MST (key).

MIS: We implemented a priority-based maximal independent sets (MIS) [17]. It calculates a score for each node by using its degree, and chooses a node having the highest score among its immediate neighbor nodes as a member of the independent set. Kimbap's MIS uses two node-property maps.

For a fair comparison, graphs are partitioned using (1) the same Cartesian vertex-cut policy [14] for CC, MSF, and MIS in Kimbap or Gluon; and (2) the same edge-cut policy for LV and LD in Kimbap or Vite (as Vite only supports edge-cuts).

6.2 Strong Scaling of Kimbap, Vite, and Gluon

Figure 9 and Figure 10 show strong scaling of Kimbap on medium size graphs with up to 16 hosts and large size graphs with up to 256 hosts respectively. They also show Vite for LV and Gluon for CC-LP. In most cases, Kimbap scales well and shows better or comparable performance to these third-party implementations; MIS needs more hosts to outperform 1 host due to higher communication-to-computation ratio.

In Figure 9a, Kimbap's LV is on average 4× faster than Vite, a hand-optimized LV implementation. Vite did not finish on large size graphs (Figure 10a) in 2.5 hours, so Kimbap is on average at least 8× faster. The main difference between Vite and Kimbap is in how they handle reductions. Vite runs an inspection phase using a single thread to construct a single shared map after communication among machines. During the execution phase, all threads concurrently perform atomic reductions on the shared-map. In contrast, Kimbap trades-off memory for performance and elides reduction conflicts among threads by using thread-local maps (Figure 7, Section 4.2). Primarily due to more (thread-local) maps, the max resident-set size (RSS) of Kimbap is on average 10% higher than that of Vite. However, Kimbap is significantly

faster than Vite in both computation and communication. The difference is higher for larger, power-law graphs due to more atomic write conflicts among threads in Vite (more details in Figure 11).

Figures 9b and 10b show strong scaling of Kimbap's LD. LD runs out-of-memory in some cases because it consumes more memory to store additional information for subclusters compared to LV. LD is on average 7× slower than LV because LD requires more edge iterations for cluster refining.

Figure 9c and 10c show the execution time of the CC algorithms in Kimbap and CC-LP in Gluon. CC-LP implementations of Kimbap and Gluon show comparable execution times as the Kimbap compiler applies communication optimizations of Gluon to adjacent-vertex operators (Section 5.2); although Kimbap uses more (thread-local) maps than Gluon, Kimbap's max RSS is similar to that of Gluon. On road-europe, CC-SCLP and CC-SV show a speedup of 14× and 2× over CC-LP on average, respectively. This is expected as road-europe has a high diameter and pointer jumping skips over multiple edges at a time. In contrast, CC-LP propagates a label only to the adjacent neighbors. CC-LP shows the fastest execution time on the power-law graphs because it propagates node properties fast through very high-degree nodes. CC-SCLP and CC-SV do not scale well as the communication cost for pointer-jumping outweighs the benefit of more compute threads.

6.3 Single Host Comparison

Table 3 compares Galois, a shared-memory (1 host) graph analytics system, with Kimbap for medium size graphs that fit in the memory of a single host. Kimbap and Galois perform comparably for LV, CC-LP, and MIS on 1 host. Kimbap on 16 hosts is on average 5×, 5×, and 3× faster than Galois on one host for LV, CC-LP, and MIS respectively.

On LD, Galois timed out at 9000s, so Kimbap is at least 4× and 14× faster on 1 host and 16 hosts respectively. Galois uses atomics to reduce the node property values in-place, so suffers from thread-conflicts for subcluster (node) property updates in LD. Kimbap avoid such conflicts entirely. Given sufficient memory, we expect Kimbap to be much faster than Galois for LD on friendster too.

On MSF and CC-SV, Galois is typically faster than Kimbap due to implementation differences. These algorithms use pointer jumping. To implement this, Galois uses atomic operations and asynchronously updates node property values. On the contrary, to update the node property values, the Kimbap implementations use BSP execution for better distributed execution efficiency. On road-europe, MSF and CC-SV in Galois are on average 28× faster than Kimbap due to the high diameter of road-europe. On friendster, MSF and CC-SV in Galois are on average 4× faster than Kimbap.

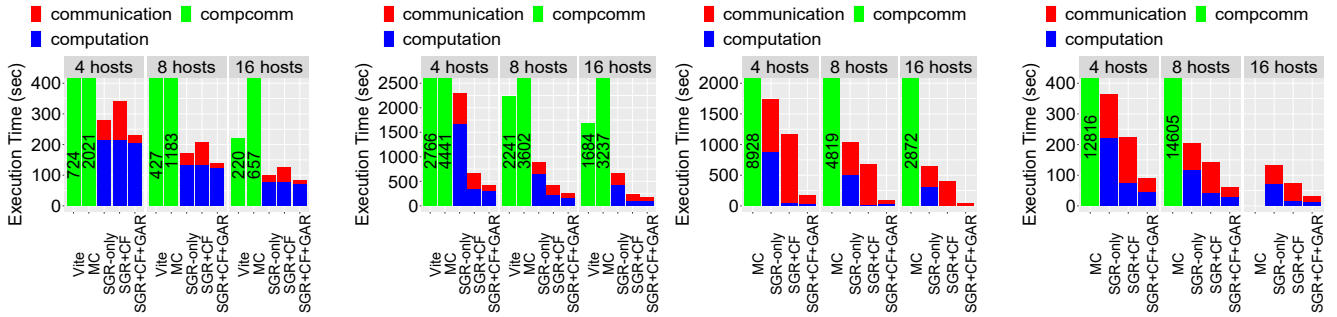
Application	Input	Galois (sec) 1 host	Kimbap (sec) 1 host	Kimbap (sec) 16 hosts
LV	road-europe	399	413	85
	friendster	839	842	140
LD	road-europe	9000	2337	638
MSF	road-europe	2	78	29
	friendster	10	56	24
CC-LP	road-europe	329	331	94
	friendster	6	5	0.8
CC-SV	road-europe	3	60	46
	friendster	17	45	22
MIS	road-europe	1	2	1
	friendster	8	9	1

Table 3. Execution time of Galois and Kimbap; LD in Galois on road-europe timed-out at 9000s; both Galois and Kimbap run OOM for LD on friendster; the best execution time is highlighted in red.

6.4 Impact of Runtime Optimizations

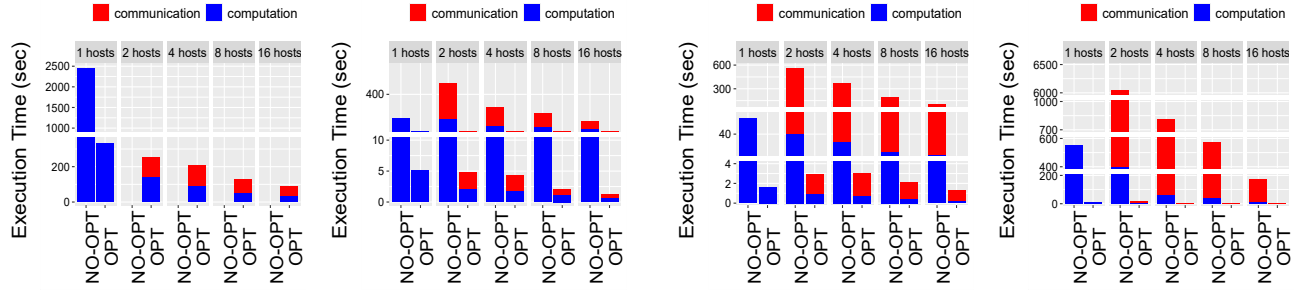
As described in Sections 4.1 and 4.2, Kimbap optimizes distributed reductions to node property maps in three ways primarily: (1) *SGR*: each host performs local reductions and these partial values are then reduced onto the owner's values, (2) *CF*: each thread performs conflict-free local reductions by leveraging thread-local maps, and (3) *GAR*: keys (nodes) are distributed such that each host owns the values (properties) for its master nodes and those values are stored in a vector while remote node properties are cached in a custom map. Pregel [58] and Vite [38] use *SGR* but they lack *CF* and *GAR*. General-purpose key-value store systems like Memcached [35] lack all three optimizations. To demonstrate the impact of each of these optimizations, we implement 4 variants of Kimbap runtime: Memcached (MC), *SGR-only*, *SGR+CF*, and *SGR+CF+GAR*. We choose *MC* as it is a publicly available in-memory key-value system which is well-known, active, and stable. All variants run the same Kimbap compiler generated programs.

In *MC*, we implemented and replaced request and reduce operations using libMemcached [2] (v1.6.17), which is a C client library for Memcached. Memcached uses modulo hashing to distribute keys. We use multiple get operations, `mget()`, for requesting master and remote values, and cache them in a custom map (similar to that for remote node properties in Figure 6). Parallelism among threads is exploited similar to that in the default Kimbap runtime. Memcached does not support a reduction operation, so we implemented it by exploiting Memcached's distributed CAS (compare-and-swap) operation; reduction operations repeat fetching canonical values from owner hosts, performing local reductions, and attempting CAS operations until they succeed. As a consequence, `ReduceSync()` is a no-op. To match Kimbap's experimental



(a) LV on road-europe **(b)** LV on friendster **(c)** CC-SV on road-europe **(d)** CC-SV on friendster

Figure 11. Execution time of Kimbap runtime variants and Vite (compcomm: computation with communication).



(a) CC-LP on road-europe **(b)** CC-LP on friendster **(c)** MIS on road-europe **(d)** MIS on friendster

Figure 12. Execution time of Kimbap with and without compiler optimizations; missing bars timed-out at 9000 seconds (NO-OPT: without compile-time optimizations, OPT: with compile-time optimizations).

setup, each host executes both a single server and a single client. We empirically choose the best configuration of 48 threads and 160GB for the store size for each server.

SGR-only uses the *SGR* optimization but similar to *MC*, it uses modulo hashing to distribute nodes and a custom map to cache both master and remote node properties. On each host, it uses a single concurrent `flat_hash_map` from *phmap* [5] (parallel-hashmap v1.33), a general-purpose shared-memory concurrent map library. All threads concurrently write partial reduction results to it. *SGR+CF* builds on top of *SGR-only* to use *CF*'s thread-local maps instead of the single concurrent map. *SGR+CF+GAR* uses partition-aware key distribution and separate representations for master and remote node properties (Figure 6); that is, it is the default node property map that leverages all the optimizations.

Figure 11 shows the execution time of these variants and Vite on the medium size graphs for LV and CC-SV (we omit the other algorithms due to lack of space). We break the execution time into computation and communication (`ReduceSync` and `RequestSync`) times, except for Vite and *MC* (computation and communication are overlapped in them).

Vite is a hand-optimized graph implementation that uses *SGR*, so it outperforms *MC* which lacks any domain-specific optimizations. Vite is however 3× slower than *SGR-only* primarily because it uses a single thread to construct a local, shared map. *SGR-only* outperforms *MC* by 11× on average. *MC* spends 71% of the execution time on loops that mainly perform distributed CAS operations and 26% of the execution time on `mget()`. This is because *MC* (1) requires more retries whereas *SGR-only* minimizes such retries by using the *SGR* optimization for distribution communication (retries are restricted to the local `flat_hash_map`); (2) requires string keys instead of Kimbap's integer keys for nodes; and (3) sends more messages per round (more metadata in volume) as opposed to one message between each pair of hosts in a round of *SGR-only*.

SGR+CF is on average 1.7× faster than *SGR-only*. *CF* improves computation time by 36% and 81% on LV and CC-SV with 6% and 12% communication overheads. *SGR-only* amortizes combining partial reduction results to the local computation phase as the partial reduction results are immediately combined to the concurrent map. For LV on road-europe,

both variants perform roughly the same on a local computation step but for LV on friendster, *SGR+CF* is on average 2.6× faster than *SGR-only*. The main reason is that most nodes in road-europe have a low degree, so there are few conflicting concurrent reductions to the same location. In contrast, the power-law graph, friendster, has many high-degree nodes, and the conflict-free design of *CF* pays off. In CC-SV, *SGR+CF* shows 24× and 3× faster computation time than *SGR-only* on both road-europe and friendster. In this case, road-europe requires more pointer-jumping as it has a high diameter, and therefore, causes more frequent concurrent reduction conflicts. *SGR+CF+GAR* is on average 3× faster than *SGR+CF* because it exploits node property locality by always caching the master node properties locally in a vector. This reduces the number of memory accesses to look up master node properties and the time needed to do so, thereby improving request, reduce, and their synchronization time.

6.5 Impact of Compile-Time Optimizations

We compare Kimbap compiler-generated codes with and without compiler optimizations for the adjacent-vertex programs, CC-LP and MIS. Figure 12 shows the results on the medium size graphs. The optimizations improve the computation time, communication time, and total runtime by 41×, 102×, and 79× on average respectively. This is a lower-bound as CC-LP without optimizations timed-out at 9000s for road-europe on more than 1 host. This shows that RequestSync elision and pinned mirrors (PM) are critical for performance.

7 Related work

Distributed graph processing frameworks. Similar to Kimbap, Pregel [58] (and Giraph [1], its open-source counterpart) provides a general vertex-centric distributed system that supports both adjacent-vertex and trans-vertex operators. Unlike Kimbap's shared-memory abstraction, Pregel uses a message passing abstraction that prevents it from decoupling computation and communication, and supporting optimizations like *GAR* and *PM* in Kimbap. Pregel also lacks Kimbap's *CF* reduction optimization and it suffers from conflicts during reductions. Kimbap supports vertex-cut policies that can help to scale out, but Pregel is restricted to edge-cut policies. Many distributed graph analytics systems [25, 25, 27, 28, 41, 47, 57, 74, 80, 82, 84, 85, 92, 93] provide a programming framework restricted to adjacent-vertex operators. Kimbap builds on top of them and includes a novel compiler and runtime optimizations that enable it to match or outperform these systems for adjacent-vertex programs.
Compilers for graph analytics. IrGL [65] generates a CUDA program for a GPU from a given sequential graph specification. Abelian [39] extends IrGL to translate shared-memory code to code that can run on distributed and heterogeneous platforms. GraphIt [16, 90] provides a high-level domain-specific language that separates algorithm and schedule specification, and includes a compiler that generates high-performance code for a shared-memory multi-core CPU or GPU. Grafs [48] provides a declarative specification language for path-based graph analytics and generates code to target existing shared-memory and distributed-memory graph processing systems. GraphIt and Grafs provide a higher-level abstraction than Kimbap. Except IrGL, all the other compilers are restricted to adjacent-vertex operators. The languages and compilation techniques in these compilers are orthogonal to that in Kimbap. Kimbap translates shared-memory code for both adjacent-vertex and trans-vertex operators into distributed-memory code.
Distributed Shared Memory (DSM) frameworks. DSM and Partitioned Global Address Space (PGAS) systems implement a shared address space in software on distributed clusters [10, 18, 20, 21, 24, 32, 33, 50, 62, 75]. DSM systems cache remote data and implement coherence in software, whereas PGAS systems do not implement caching of remote data. While DSM and PGAS systems support trans-vertex operators, they do so using general-purpose mechanisms, which are not as efficient as Kimbap's domain-specific optimizations. In particular, the *SGR* and *CF* reductions are optimized for highly concurrent sparse reductions, and this cannot easily be emulated in these systems.
Distributed key-value store frameworks. Distributed key-value store systems provide a data storage and interfaces to store, manage, read, and write key-value pairs across distributed-memory machines. Trans-vertex operators can be implemented in these systems, but many systems [19, 22, 23, 42, 51, 83] are out-of-core systems and optimize I/O operations, which is orthogonal to our work as we focus on in-memory workloads. LinkBench [11] is designed to process data in social networks and it supports graph morph operations such as node/edge addition/deletion, whereas Kimbap focuses on read/reduce operations on node-property maps. Redis [4] and Memcached [35] are widely used general-purpose in-memory key-value store systems. Kimbap includes a custom in-memory key-value store for node-property maps that leverage domain-specific optimizations such as *GAR*, *CF* reductions, and *SGR*. Our evaluation shows the impact of these optimizations compared to Memcached.
Hand-optimized distributed graph analytics. Behnezhad et al. [12] implement minimum spanning forest by exploiting a generic key-value store system [22]. It caches graph topologies in the key-value store and accesses arbitrary node properties (it does not use reductions on the key-value store). In contrast, Kimbap only uses node property maps and leverages reduction operations that are more lightweight and efficient. Vite [38] and SCLP [78] implement new Louvain clustering and connected components algorithms respectively on distributed-memory systems with algorithm-level optimizations. These hand-optimized implementations require the programmer to handle distributed communication.

8 Conclusion

This paper presented Kimbap, a distributed graph analytics system optimized for both adjacent-vertex and trans-vertex operators. A novel distributed node-property map implements conflict-free reductions and efficient reads by leveraging partitioning information. The compiler hides the complexity of the distributed-memory from application programmers and permits generation of specialized code for adjacent-vertex programs. In this way, the Kimbap system enables application programmers to write programs at a high level of abstraction while outperforming state-of-the-art, hand-optimized programs even for complex algorithms like Louvain clustering.

9 Acknowledgements

This material is based upon work partially supported by the U.S. Department of Energy, National Nuclear Security Administration Award Number DE-NA0003969, and the Army Research Office and IARPA under Contract No. W911NF-22-C-0085. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the DOE, Army Research Office, and IARPA. The Texas Advanced Computing Centers (TACC) and the NSF provided scheduling unit allocation on the TACC Stampede2 system for the evaluation. We thank the anonymous reviewers and our shepherd, Fredrik Kjolstad, for their many suggestions in improving our paper.

References

[1] Apache Giraph. http://giraph.apache.org.
[2] libmemcached. https://libmemcached.org.
[3] Neo4j. https://neo4j.com/.
[4] Redis. http://redis.io/.
[5] The Parallel Hashmap. https://github.com/greg7mdp/parallel-hashmap.
[6] Tigergraph. https://www.tigergraph.com/.
[7] Texas Advanced Computing Center (TACC), The University of Texas at Austin, 2018.
[8] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, and tools*. Addison Wesley, 1986.
[9] Vahid Khalilpour Akram and Onur Ugurlu. A localized distributed algorithm for vertex cover problem. *Journal of Computational Science*, 58, 2022.
[10] Cristiana Amza, Alan L Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: shared memory computing on networks of workstations. *Computer*, pages 18–28, 1996.
[11] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. Linkbench: A database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 1185–1196. Association for Computing Machinery, 2013.
[12] Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Lacki, Vahab Mirrokni, and Warren Schudy. Parallel graph algorithms in constant adaptive rounds: Theory meets practice. *Proceedings of the VLDB Endowment*, 13(13).
[13] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.
[14] Erik G Boman, Karen D Devine, and Sivasankaran Rajamanickam. Scalable matrix computations on large scale-free graphs using 2D graph partitioning. In *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, 2013.
[15] Otakar Boruvka. Contribution to the solution of a problem of economical construction of electrical networks. *Elektronickỳ Obzor*, 15:153–154, 1926.
[16] Ajay Brahmakshatriya, Yunming Zhang, Changwan Hong, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Compiling graph applications for gpu s with graphit. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 248–261, 2021.
[17] Martin Burtscher, Sindhu Devale, Sahar Azimi, Jayadharini Jaiganesh, and Evan Powers. A high-quality and fast maximal independent set implementation for gpus. *ACM Transactions on Parallel Computing (TOPC)*, pages 1–27, 2018.
[18] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient distributed memory management with rdma and caching. *Proceedings of the VLDB Endowment*, pages 1604–1617, 2018.
[19] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.
[20] John B Carter. Design of the munin distributed shared memory system. *Journal of Parallel and Distributed Computing*, pages 219–227, 1995.
[21] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, pages 291–312, 2007.
[22] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. *ACM Sigplan Notices*, 45(6):363–375, 2010.
[23] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. Faster: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*, pages 275–290, 2018.
[24] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *ACM SIGPLAN NOTICES*, pages 519–538, 2005.
[25] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)*, 2019.
[26] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.
[27] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. Gluon: A Communication-optimizing Substrate for Distributed Heterogeneous Graph Analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 752–768. ACM, 2018.
[28] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Vishwesh Jatala, Keshav Pingali, V Krishna Nandivada, Hoang-Vu Dang, and Marc Snir. Gluon-async: A bulk-asynchronous system for distributed and heterogeneous graph analytics. In *PACT'19*, pages 15–28. IEEE, 2019.

[29] Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. Phoenix: A substrate for resilient distributed graph analytics. In *ASPLOS'19*, pages 615–630, 2019.

[30] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor Lee. Tigergraph: A native mpp graph database. *arXiv preprint arXiv:1901.08248*, 2019.

[31] Irit Dinur and Samuel Safra. On the hardness of approximating minimum vertex cover. *Annals of mathematics*, pages 439–485, 2005.

[32] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. {FaRM}: Fast remote memory. In *NSDI 14*, pages 401–414, 2014.

[33] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: distributed shared memory programming*. John Wiley & Sons, 2005.

[34] Scott Emmons, Stephen Kobourov, Mike Gallant, and Katy Börner. Analysis of network clustering algorithms and cluster quality metrics at scale. *PloS one*, 11(7):e0159161, 2016.

[35] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[36] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 international conference on management of data*, pages 1433–1445, 2018.

[37] Karlsruher Institut fur Technologie. OSM-Europe. https://i11www.iti.kit.edu/resources/roadgraphs.php, 2014.

[38] Sayan Ghosh, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, Hao Lu, Daniel Chavarrià-Miranda, Arif Khan, and Assefaw Gebremedhin. Distributed louvain algorithm for graph community detection. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 885–895, 2018.

[39] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Andrew Lenharth, and Keshav Pingali. Abelian: A Compiler for Graph Analytics on Distributed, Heterogeneous Platforms. In *Euro-Par 2018: Parallel Processing*, 2018.

[40] Gurbinder Gill, Roshan Dathathri, Loc Hoang, and Keshav Pingali. A Study of Partitioning Policies for Graph Analytics on Large-scale Distributed Platforms. volume 12 of *PVLDB*, 2018.

[41] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. {PowerGraph}: Distributed {Graph-Parallel} computation on natural graphs. In *OSDI'12*, pages 17–30, 2012.

[42] Harshit Gupta and Umakishore Ramachandran. Fogstore: A geo-distributed key-value store guaranteeing low latency for strongly consistent access. In *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*, pages 148–159, 2018.

[43] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), 2008.

[44] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. Ordered vs unordered: a comparison of parallelism and work-efficiency in irregular algorithms. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 3–12, New York, NY, USA, 2011. ACM.

[45] Jonathan Haynes and Igor Perisic. Mapping search relevance to social networks. In *Proceedings of the 3rd Workshop on Social Network Mining and Analysis*, pages 1–7, 2009.

[46] Loc Hoang, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. CuSP: A Customizable Streaming Edge Partitioner for Distributed Graph Analytics. In *Proceedings of the 33rd IEEE International Parallel and Distributed Processing Symposium*, IPDPS 2019, 2019.

[47] Imranul Hoque and Indranil Gupta. Lfgraph: Simple and fast distributed graph analytics. TRIOS '13, New York, NY, USA, 2013. Association for Computing Machinery.

[48] Farzin Houshmand, Mohsen Lesani, and Keval Vora. Grafs: Declarative graph analytics. (ICFP), 2021.

[49] Glen Jeh and Jennifer Widom. Simrank: a measure of structural-context similarity. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 538–543, 2002.

[50] Stefanos Kaxiras, David Klaftenegger, Magnus Norgren, Alberto Ros, and Konstantinos Sagonas. Turning centralized coherence and distributed critical-section execution on their head: A new approach for scalable distributed shared memory. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, 2015.

[51] Avinash Lakshman and Prashant Malik. Cassandra: structured storage system on a p2p network. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 5–5, 2009.

[52] Youngho Lee, Yubin Lee, Jeong Seong, Ana Stanescu, and Chul Sue Hwang. A comparison of network clustering algorithms in keyword network analysis: A case study with geography conference presentations. *International Journal of Geospatial and Environmental Research*, 7(3):1, 2020.

[53] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. Parallel Graph Analytics. *Commun. ACM*, 59(5):78–87, April 2016.

[54] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data, 2014.

[55] Aristidis Likas, Nikos Vlassis, and Jakob J Verbeek. The global k-means clustering algorithm. *Pattern recognition*, 36(2):451–461, 2003.

[56] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, apr 2012.

[57] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. {NeuGraph}: Parallel deep neural network computation on large graphs. In *USENIX ATC 19*, pages 443–458, 2019.

[58] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[59] Artur Mariano, Alberto Proenca, and Cristiano Da Silva Sousa. A generic and highly efficient parallel variant of boruvka's algorithm. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 610–617. IEEE, 2015.

[60] David Meunier, Renaud Lambiotte, Alex Fornito, Karen Ersche, and Edward T Bullmore. Hierarchical modularity in human brain functional networks. *Frontiers in neuroinformatics*, 3:571, 2009.

[61] Robert Meusel, Oliver Lehmberg, Christian Bizer, and Sebastiano Vigna. Web data commons-hyperlink graphs. *URL http://webdatacommons. org/hyperlinkgraph/index. html*, 2012.

[62] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. {Latency-Tolerant} software distributed shared memory. In *USENIX ATC'15*, pages 291–305, 2015.

[63] Mark EJ Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

[64] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM.

[65] Sreepathi Pai and Keshav Pingali. A Compiler for Throughput Optimization of Graph Algorithms on GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 1–19, New York, NY, USA, 2016. ACM.

[66] Anthony Papantonakis and Peter J. H. King. Syntax and semantics of gql, a graphical query language. *Journal of Visual Languages & Computing*, 6(1):3–25, 1995.

[67] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew

Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The TAO of parallelism in algorithms. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, PLDI '11, pages 12–25, 2011.

[68] Robert Clay Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, pages 1389–1401, 1957.

[69] The Lemur Project. The ClueWeb12 Dataset, 2013.

[70] Dimitrios Prountzos and Keshav Pingali. Betweenness centrality: algorithms and implementations. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13, pages 35–46, New York, NY, USA, 2013. ACM.

[71] Troy Raeder and Nitesh V Chawla. Market basket analysis with networks. *Social network analysis and mining*, 1:97–113, 2011.

[72] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E*, 76(3):036106, 2007.

[73] Anne Rogers and Keshav Pingali. Compiling for distributed memory architectures. *IEEE Trans. Parallel Distrib. Syst.*, 5(3):281–298, 1994.

[74] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *SOSP'15*, pages 410—-424, 2015.

[75] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, page 297–306, 1994.

[76] Yossi Shiloach and Uzi Vishkin. An o(logn) parallel connectivity algorithm. *Journal of Algorithms*, 1982.

[77] Dan Stanzione, Bill Barth, Niall Gaffney, Kelly Gaither, Chris Hempel, Tommy Minyard, S. Mehringer, Eric Wernert, H. Tufo, D. Panda, and P. Teller. Stampede 2: The Evolution of an XSEDE Supercomputer. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, PEARC, pages 15:1–15:8. ACM, 2017.

[78] Stergios Stergiou, Dipen Rughwani, and Kostas Tsioutsiouliklis. Shortcutting label propagation for distributed connected components. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, pages 540–546, 2018.

[79] Vincent A Traag, Ludo Waltman, and Nees Jan Van Eck. From louvain to leiden: guaranteeing well-connected communities. *Scientific reports*, 2019.

[80] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. Aspire: Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based dsm. *ACM SIGPLAN Notices*, 49(10):861–878, 2014.

[81] Keval Vora, Chen Tian, Rajiv Gupta, and Ziang Hu. Coral: Confined recovery in distributed asynchronous graph processing. *ACM SIGARCH Computer Architecture News*, pages 223–236, 2017.

[82] Keval Vora, Chen Tian, Rajiv Gupta, and Ziang Hu. Coral: Confined recovery in distributed asynchronous graph processing. *ACM SIGARCH Computer Architecture News*, 45(1):223–236, 2017.

[83] Chenggang Wu, Jose M Faleiro, Yihan Lin, and Joseph M Hellerstein. Anna: A kvs for any scale. *IEEE Transactions on Knowledge and Data Engineering*, 33(2):344–358, 2019.

[84] Wencong Xiao, Jilong Xue, Youshan Miao, Zhen Li, Cheng Chen, Ming Wu, Wei Li, and Lidong Zhou. Tux^2: Distributed graph computation for machine learning. In *NSDI'17*, pages 669–682, 2017.

[85] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. Sync or async: Time to fuse for distributed graph-parallel computation. *ACM SIGPLAN Notices*, 50(8):194–204, 2015.

[86] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. Pregel algorithms for graph connectivity problems with performance guarantees. 2014.

[87] Zhao Yang, René Algesheimer, and Claudio J Tessone. A comparative analysis of community detection algorithms on artificial networks. *Scientific reports*, 6(1):30750, 2016.

[88] Hao Yin, Austin R Benson, and Jure Leskovec. The local closure coefficient: A new perspective on network clustering. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, pages 303–311, 2019.

[89] Lin Zhang, Xinhai Liu, Frizo Janssens, Liming Liang, and Wolfgang Glänzel. Subject clustering analysis based on isi category classification. *Journal of Informetrics*, 4(2):185–193, 2010.

[90] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph dsl. (OOPSLA), 2018.

[91] Zhongying Zhao, Shaoqiang Zheng, Chao Li, Jinqing Sun, Liang Chang, and Francisco Chiclana. A comparative study on community detection methods in complex networks. *Journal of Intelligent & Fuzzy Systems*, 35(1):1077–1086, 2018.

[92] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A {Computation-Centric} distributed graph processing system. In *OSDI'16*, pages 301–316, 2016.

[93] Youwei Zhuo, Jingji Chen, Qinyi Luo, Yanzhi Wang, Hailong Yang, Depei Qian, and Xuehai Qian. Symplegraph: distributed graph processing with precise loop-carried dependency guarantee. In *PLDI'20*, pages 592–607, 2020.