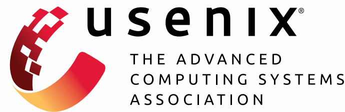

Gemini: A Computation-Centric Distributed Graph Processing System

Xiaowei Zhu, Wenguang Chen, and Weimin Zheng, ***Tsinghua University;***
Xiaosong Ma, ***Hamad Bin Khalifa University***

https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhu

This paper is included in the Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16).

November 2–4, 2016 • Savannah, GA, USA

ISBN 978-1-931971-33-1

Open access to the Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

Gemini: A Computation-Centric Distributed Graph Processing System

Xiaowei Zhu[1], Wenguang Chen[1,2,*], Weimin Zheng[1], and Xiaosong Ma[3]

[1]Department of Computer Science and Technology (TNLIST), Tsinghua University
[2]Technology Innovation Center at Yinzhou,
Yangtze Delta Region Institute of Tsinghua University, Zhejiang
[3]Qatar Computing Research Institute, Hamad Bin Khalifa University

Abstract

Traditionally distributed graph processing systems have largely focused on scalability through the optimizations of inter-node communication and load balance. However, they often deliver unsatisfactory overall processing efficiency compared with shared-memory graph computing frameworks. We analyze the behavior of several graph-parallel systems and find that the added overhead for achieving scalability becomes a major limiting factor for efficiency, especially with modern multi-core processors and high-speed interconnection networks.

Based on our observations, we present Gemini, a distributed graph processing system that applies multiple optimizations targeting computation performance to *build scalability on top of efficiency*. Gemini adopts (1) a sparse-dense signal-slot abstraction to extend the hybrid push-pull computation model from shared-memory to distributed scenarios, (2) a chunk-based partitioning scheme enabling low-overhead scaling out designs and locality-preserving vertex accesses, (3) a dual representation scheme to compress accesses to vertex indices, (4) NUMA-aware sub-partitioning for efficient intra-node memory accesses, plus (5) locality-aware chunking and fine-grained work-stealing for improving both inter-node and intra-node load balance, respectively. Our evaluation on an 8-node high-performance cluster (using five widely used graph applications and five real-world graphs) shows that Gemini significantly outperforms all well-known existing distributed graph processing systems, delivering up to 39.8× (from 8.91×) improvement over the fastest among them.

1 Introduction

Graph processing is gaining increasing attentions in both academic and industrial communities. With the magnitude of graph data growing rapidly, many specialized distributed systems [3, 12, 16, 17, 30, 32, 41, 45] have been proposed to process large-scale graphs.

While these systems are able to take advantage of multiple machines to achieve scalability, their performance is often unsatisfactory compared with state-of-the-art shared-memory counterparts [36, 47, 49, 57]. Further, a recent study [33] shows that an optimized single-thread implementation is able to outperform many distributed systems using many more cores. Our hands-on experiments and performance analysis reveal several types of design and implementation deficiencies that lead to loss of performance (details in Section 2).

Based on the performance measurement and code examinations, we come to recognize that traditional distributed graph-parallel systems do not fit in today's powerful multi-core cluster nodes and fast-speed networks. To achieve better overall performance, one needs to focus on the performance of both computation and communication components, compressing the computation time aggressively while hiding the communication cost, rather than focusing primarily on minimizing communication volume, as seen in multiple existing systems' design.

To bridge the gap between efficient shared-memory and scalable distributed systems, we present Gemini, a distributed graph processing system that builds *scalability* on top of *efficiency*. More specifically, the main contributions of this work are summarized as follows:

- We perform detailed analysis of several existing shared-memory and distributed graph-parallel systems and identify multiple design pitfalls.
- We recognize that efficient and scalable distributed graph processing involves intricate interplay between the properties of the application, the underlying system, and the input graph. In response, we explore adaptive runtime choices, such as a density-aware dual-mode processing scheme and multiple locality-aware data distribution and load balancing mechanisms. The result is a system that can deliver competitive performance on a range of system

*Corresponding author (cwg@tsinghua.edu.cn).

scales, from multi-core to multi-node platforms.

- We identify a simple yet surprisingly effective chunk-based graph partitioning scheme, which facilitates exploitation of natural locality in input graphs and enables seamless hierarchical refinement. We present multiple optimizations enabled by this new partitioning approach.
- We evaluate our Gemini prototype with extensive experiments and compared it with five state-of-the-art systems. Experiments with five applications on five real-world graphs show that Gemini significantly outperforms existing distributed implementations, delivering up to 39.8× (from 8.91×) improvement over the fastest among them. We collect detailed measurement for performance analysis and validating internal design choices.

2 Motivation

While state-of-the-art shared-memory graph processing systems are able to process graphs quite efficiently, the lack of scalability makes them fail to handle graphs that do not fit in the memory of a single machine. On the other hand, while existing distributed solutions can scale graph processing to larger magnitudes than their shared-memory counterparts, their performance and cost efficiencies are often unsatisfactory [33, 59, 60].

To study the performance loss, we profiled several representative graph-parallel systems, including Ligra [47], Galois [36], PowerGraph [16], PowerLyra [12], as well as the optimized single-thread implementation proposed in the COST paper [33] for reference. We set up experiments on an 8-node high-performance cluster interconnected with Infiniband EDR network (with up to 100Gbps bandwidth), each node containing two Intel Xeon E5-2670 v3 CPUs (12 cores and 30MB L3 cache per CPU) and 128 GB DRAM. We ran 20 iterations of PageRank [38] on the *twitter-2010* [28] graph, a test case commonly used for evaluating graph-parallel systems.

Cores	1	24 × 1		24 × 8	
System	OST	Ligra	Galois	PowerG.	PowerL.
Runtime (s)	99.9	21.9	19.3	40.3	26.9
Instructions	525G	496G	482G	7.15T	6.06T
Mem. Ref.	15.8G	32.3G	23.4G	95.8G	87.2G
Comm. (GB)	-	-	-	115	38.1
IPC	1.71	0.408	0.414	0.500	0.655
LLC Miss	8.77%	43.9%	49.7%	71.0%	54.9%
CPU Util.	100%	91.7%	96.8%	65.5%	68.4%

Table 1: Sample performance analysis of existing systems (20 iterations of PageRank on *twitter-2010*). OST refers to the optimized single-thread implementation.

Table 1 gives detailed performance metrics for the five targeted systems. Overall, systems lose efficiency as we move from single-thread to shared memory, then to distributed implementations. Though this is to be expected with communication/synchronization overhead, load balance issues, and in general higher software complexities, the large span in almost all measurement categories across alternative systems indicates a large room for improvement.

As seen from the profiling results, the network is far from saturated (*e.g.*, lower than 3Gbps average aggregate bandwidth usage with PowerGraph). Computation, rather than communication, appears to be the actual bottleneck of evaluated distributed systems, which echoes recent findings on distributed data analytics frameworks [37]. Compared with their shared-memory counterparts, they have significantly more instructions and memory references, poorer access localities, and lower multi-core utilization. We further dig into the code and find that such inefficiency comes from multiple sources, such as (1) the use of hash maps to convert vertex IDs between global and local states, (2) the maintenance of vertex replicas, (3) the communication-bound apply phase in the GAS abstraction [16], and (4) the lack of dynamic scheduling. They either enlarge the working set, producing more instructions and memory references, or prevent the full utilization of multi-core CPUs.

We argue that many of the above side-effects could be avoided when designing distributed graph-parallel systems, by building *scalability* on top of *efficiency*, instead of focusing on the former in the first place. The subsequent distributed system design should pay close attention to the computation overhead of cross-node operation over today's high-speed interconnect, as well as the local computation efficiency on partitioned graphs.

To this end, we adapt Ligra's hybrid push-pull computation model to a distributed form, which facilitates efficient vertex-centric data update and message passing. A chunk-based partitioning scheme is adopted, allowing low-overhead graph distribution as well as recursive application at multiple system levels. We further deploy multiple optimizations to aggressively compress the computation time. Finally, we design a co-scheduling mechanism to overlap computation and inter-node communication tasks.

3 Gemini Graph Processing Abstraction

Viewing modern clusters as small or moderate number of nodes interconnected with fast networks similar to a shared-memory multi-core machine, Gemini adopts a graph processing abstraction that enables a smooth extension of state-of-the-art single-node graph computation models to cluster environments.

Before getting to details, let us first give the targeted graph processing context. Like assumed in many graph-

parallel systems or frameworks, single-node [47, 59, 60] or distributed [11, 18, 23, 50] alike, a graph processing problem updates information stored in vertices, while edges are viewed as immutable objects. Also, like common systems, Gemini processes both directed and undirected graphs, though the latter could be converted to directed ones by replacing each undirected edge with a pair of directed edges. The rest of our discussion therefore assumes all edges are directed.

For a common graph processing application, the processing is done by propagating vertex updates along the edges, until the graph state converges or a given number of iterations are completed. Vertices with ongoing updates are called *active vertices*, whose outgoing edges collectively form the *active edge set* for processing.

3.1 Dual Update Propagation Model

At a given time during graph processing, the active edge set may be *dense* or *sparse*, typically determined by its size (total number of outgoing edges from active vertices) relative to $|E|$, the total number of edges. For example, the active edge set of the *CC* (connected components) application is dense in the first few iterations, and gets increasingly sparse as more vertices receive their final labels. *SSSP* (single-source shortest paths), on the other hand, starts from a very sparse edge set, getting denser as more vertices become activated by their in-neighbors, and sparse again when the algorithm approaches the final convergence.

State-of-the-art shared-memory graph processing systems [4, 36, 47, 57] have recognized that different active edge set densities call for different update propagation models. More specifically, sparse active edge sets prefer the *push* model (where updates are passed to neighboring vertices along outgoing edges), as the system only traverses outgoing edges of *active* vertices where new updates are made. In contrast, dense active edge sets benefit more from the *pull* model (where each vertex's update is done by collecting states of neighboring vertices along incoming edges), as this significantly reduces the contention in updating vertex states via locks or atomic operations.

While Ligra [47] proposed the adaptive switch between these two modes according to the density of an active edge set in a shared-memory machine (with the default threshold $|E|/20$, which Gemini follows), here we explore the feasibility of extending such design to distributed systems. The major difference is that a graph will be partitioned and distributed across different nodes, where information and updates are shared using explicit message passing. To this end, Gemini uses the master-mirror notion as in PowerGraph [16]: each vertex is assigned to (*owned by*) one partition, where it is a *master* vertex, as the primary copy maintaining vertex state data. The same vertex may also have replicas, called *mirrors*, on each node/partition that owns at least one of its neighbors. A pair of directed edges will be created between each master-mirror pair, though only one of them will be used in either propagation mode. Note that unlike in PowerGraph, mirrors in Gemini act like placeholders only for update propagation and do not hold actual data.

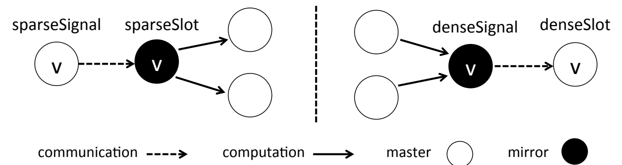

Figure 1: The sparse-dense signal-slot model

With replicated vertices, Gemini adopts a sparse-dense dual engine design, using a signal-slot abstraction to decouple the propagation of vertex states (communication) from the processing of edges (computation). Borrowed but slightly different from in the Qt software framework [1], *signals* and *slots* denote user-defined vertex-centric functions describing message sending and receiving behaviors, respectively. Computation and communication are handled differentially in the two modes, as illustrated in Figure 1. In the sparse (*push*) mode, each master first sends messages containing latest vertex states to its mirrors via `sparseSignal`, who in turn update their neighbors through outgoing edges via `sparseSlot`. In the dense (*pull*) mode, each mirror first performs local computation based on states of neighboring vertices through incoming edges, then sends an update message containing the result to its master via `denseSignal`, who subsequently updates its own state appropriately via `denseSlot`.

An interesting feature of the proposed abstraction is that *message combining* [32] is automatically enabled. Only one message per active master-mirror pair of each vertex is needed, lowering the number of messages from $O(|E|)$ to $O(|V|)$. This also allows computation to be performed locally to aggregate outgoing updates without adopting an additional "combining pass", which is necessary in many Pregel-like systems [3, 32, 44].

3.2 Gemini API

Gemini adopts an API design (Figure 2) similar to those presented by shared-memory systems [47, 57]. Data and computation distribution details are hidden from users. A graph is described in its entirety with a type `E` for edge data, and several user-defined vertex arrays. A compact `VertexSet` data structure (internally implemented with

```
class Graph<E> {
  VertexID vertices;
  EdgeID edges;
  VertexID [] outDegree;
  VertexID [] inDegree;
  def allocVertexArray<V>() -> V [];
  def allocVertexSet() -> VertexSet;
  def processVertices<A> (
    work: (VertexID) -> A,
    active: VertexSet,
    reduce: (A, A) -> A,
  ) -> A;
  def processEdges<A, M> (
    sparseSignal: (VertexID) -> void,
    sparseSlot: (VertexID, M, OutEdgeIterator<E>) -> A,
    denseSignal: (VertexID, InEdgeIterator<E>) -> void,
    denseSlot: (VertexID, M) -> A,
    reduce: (A, A) -> A,
    active: VertexSet
  ) -> A;
  def emit<M> (recipient: VertexID, message: M) -> void;
};
```

Figure 2: Core Gemini API

bitmaps) is provided for efficient representation of a vertex subset, *e.g.*, the active vertex set.

The only major Gemini APIs for users to provide custom codes are those specifying computation tasks, namely the `processVertices` and `processEdges` collections. For the rest of this section we illustrate the semantics of these user-defined functions using *CC* as an example.

```
Graph<empty> g (...); // load a graph from the file system
VertexSet activeCurr = g.allocVertexSet();
VertexSet activeNext = g.allocVertexSet();
activeCurr.fill(); // add all vertices to the set
VertexID [] label = g.allocVertexArray <VertexID> ();
def add (VertexID a, VertexID b) : VertexID {
  return a + b;
}
def initialize (VertexID v) : VertexID {
  label[v] = v;
  return 1;
}
VertexID activated = g.processVertices <VertexID> (
  initialize,
  activeCurr
);
```

Figure 3: Definitions and initialization for *CC*

As illustrated in Figure 3, we first create active vertex sets for the current/next iteration, define the `label` vertex array, and initialize the latter with own vertex IDs through a `processVertices` call.

A classic iterative label propagation method is then used to compute the connected components. Figure 4 gives the per-iteration update logic defined in two `signal-slot` pairs. In the sparse mode, every active vertex first broadcasts its current label from the master to its mirrors, including the master itself (`CCSparseSignal`). When a mirror receives the label, it iterates over its *local* outgoing edges and updates the vertex states of its neighbors (`CCSparseSlot`). In the dense mode, each vertex (both masters and mirrors, active or inactive) first iterates over its *local* incoming edges and sends the smallest label from the neighborhood to its master (`CCDenseSignal`). The master updates its own vertex state, upon receiving a label smaller than its current one (`CCDenseSlot`). The number of overall vertex activations in the current itera-

```
def CCSparseSignal (VertexID v) {
  g.emit(v, label[v]);
}
def CCSparseSlot (VertexID v, VertexID msg, OEI iter) : VertexID
{
  VertexID activated = 0;
  while (iter.hasNext()) {
    VertexID dst = iter.next().neighbour;
    if (msg < label[dst] && atomicWriteMin(label[dst], msg)) {
      activeNext.add(dst); // add 'dst' to the next frontier
      activated += 1;
    }
  }
  return activated;
}
def CCDenseSignal (VertexID v, IEI iter) : void {
  VertexID msg = v;
  while (iter.hasNext()) {
    VertexID src = iter.next().neighbour;
    msg = msg < label[src] ? msg : label[src];
  }
  if (msg < v) g.emit(v, msg);
}
def CCDenseSlot (VertexID v, VertexID msg) : VertexID {
  if (msg < label[v] && atomicWriteMin(label[v], msg)) {
    activeNext.add(v); // add 'v' to the next frontier
    return 1;
  }
  else return 0;
}
while (activated>0) {
  activeNext.clear(); // make an empty vertex set
  activated = g.processEdges <VertexID, VertexID> (
    CCSparseSignal,
    CCSparseSlot,
    CCDenseSignal,
    CCDenseSlot,
    activeCurr,
    add
  );
  swap(activeCurr, activeNext);
}
```

Figure 4: Iterative label propagation for *CC*. `OEI` and `IEI` are edge iterators for outgoing edges and incoming edges respectively. `atomicWriteMin`(a, b) atomically assigns b to a if $b < a$. `swap`(a, b) exchanges a and b.

tion is collected, via aggregating the `slot` function return values using `add`, to determine whether convergence has been reached.

Note that in Gemini, graph partitioning stops at the socket level. Cores on the same socket do not communicate via message passing, but directly perform updates on shared graph data. Therefore, as shown in the example, atomic operations are used in *slots* to ensure that vertex states are updated properly.

Not all user-defined functions are mandatory, *e.g.*, `reduce` is not necessary where there is no aggregation. Also, Gemini's dual mode processing is optional: users may choose to supply only the sparse or dense mode algorithm implementation, especially if it is known that staying at one mode delivers adequate performance (such as dense mode for PageRank), or when the memory is only able to hold edges in one direction.

4 Distributed Graph Representation

While Gemini's computation model presents users with a unified logical view of the entire graph, when deployed on a high-performance cluster, the actual graph has to be partitioned and distributed internally to exploit parallelism. A number of partitioning methods have been proposed, including both *vertex-centric* [30, 32, 48] and *edge-centric* (*aka. vertex-cut*) [8, 12, 16, 24, 39] solutions. Vertex-centric solutions enable a centralized computation model, where vertices are evenly assigned to partitions, along with their associated data, such as vertex states and adjacent edges. Edge-centric solutions, on the other hand, evenly assign edges to partitions and replicate vertices accordingly.

However, as profiling results in Section 2 demonstrated, prior studies focused on partitioning for load balance and communication optimizations, without paying enough attention on the resulted system complexity and the implication of partitioning design choices on the efficiency of computation.

To achieve scalability while maintaining efficiency, we propose a lightweight, chunk-based, multi-level partitioning scheme. We present several design choices regarding graph partitioning and internal representation that aim at improving the *computation performance* in distributed graph processing.

4.1 Chunk-Based Partitioning

The inspiration of Gemini's chunk-based partitioning comes from the fact that many large-scale real-world graphs often possess natural locality, with crawling being the common way to collect them in the first place. Adjacent vertices likely to be stored close to each other. Partitioning the vertex set into contiguous chunks could effectively preserve such locality. For example, in typical web graphs the lexicographical URL ordering guarantees that most of the edges connect two vertices close to each other (in vertex ID) [7]; in the Facebook friendship network, most of the links are close in geo-locations [52]. When locality happens to be lost in the input, there also exist effective and affordable methods to "recover" locality from the topological structure [2, 5], bringing the benefit of chunk-based partitioning for potentially repeated graph processing at a one-time pre-processing cost.

On a p-node cluster, a given global graph $G = (V, E)$ will be partitioned into p subgraphs $G_i = (V_i', E_i)$ (i from 0 to $p-1$), where V_i' and E_i are the vertex subset and the edge subset on the ith partition, respectively. To differentiate master vertices from others, we denote V_i to be the owned vertex subset on the ith partition.

Gemini partitions G using a simple chunk-based scheme, dividing V into p contiguous vertex chunks $(V_0, V_1, ..., V_{p-1})$, whose sizes are determined by additional optimizations discussed later in this section. Further, we use E_i^S and E_i^D to represent the outgoing and incoming edge set of partition i, used in the sparse and dense mode respectively. Each chunk (V_i) is assigned to one cluster node, which owns all vertices in this chunk. Edges are then assigned by the following rules:

$$E_i^S = \{(src, dst, value) \in E | dst \in V_i\}$$

$$E_i^D = \{(src, dst, value) \in E | src \in V_i\}$$

where src, dst, and $value$ represent an edge's source vertex, destination vertex, and edge value, respectively. In other words, for the ith partition, the outgoing edge set E_i^S contains edges *destined to* its owned vertices V_i, while the incoming edge set E_i^D contains edges *sourced from* V_i.

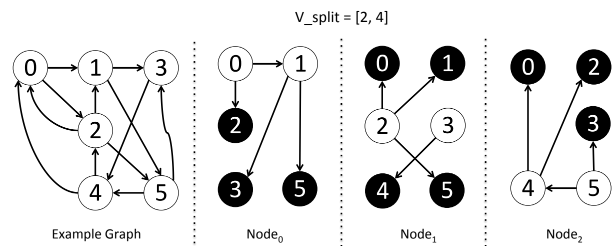

Figure 5: An example of chunk-based partitioning (dense mode), where the ID-ordered vertex array is split into three chunks $\{0,1\}, \{2,3\}, \{4,5\}$. Again black and white vertices denote mirrors and masters respectively.

Figure 5 gives an example of chunk-based partitioning, showing the vertex set on three nodes, with their corresponding dense mode edge sets. Here mirrors are created for all remote vertices that local masters have *out* edges to. These mirrors will "pull" local neighboring

states to update their remote masters. The sparse mode edge sets are similar and omitted due to space limit.

With chunk-based partitioning, Gemini achieves scalability with little overhead. The contiguous partitioning enables effortless vertex affiliation queries, by simply checking partition boundaries. The contiguous feature within each chunk also simplifies vertex data representation: only the owned parts of vertex arrays are actually touched and allocated in contiguous memory pages on each node. Therefore, the memory footprint is well controlled and no vertex ID conversions are needed to compress the space consumption of vertex states.

As accesses to neighboring states generate random accesses in both push and pull modes, vertex access locality is often found to be performance-critical. Chunk-based partitioning naturally preserves the vertex access locality, which tends to be lost when random-based distribution is used. Moreover, random accesses to vertex states all falls into the owned chunk V_i rather than V or V_i'. Gemini can then benefit from chunk-based partitioning when the system scales out, where random accesses could be handled more efficiently as the chunk size decreases.

Such lightweight chunk-based partitioning does sacrifice balanced edge distribution or minimized cut edge set, but compensates for such deficiency by (1) low-overhead scaling out designs, (2) preserved memory access localities, and (3) additional load balancing and task scheduling optimizations to be presented later in the paper.

4.2 Dual-Mode Edge Representation

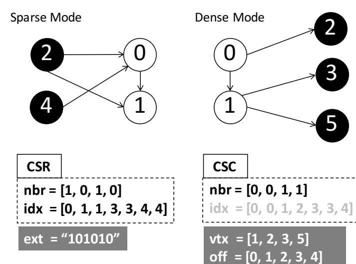

Figure 6: Sample representation of sparse/dense mode edges using CSR/CSC, with Gemini enhancement highlighted

Gemini organizes outgoing edges in the *Compressed Sparse Row* (*CSR*) and incoming ones in the *Compressed Sparse Column* (*CSC*) format. Both are compact sparse matrix data structures commonly used in graph systems, facilitating efficient vertex-centric sequential edge access. Figure 6 illustrates the graph partition on cluster node 0 from the sample graph in Figure 5 and its CSR/CSC representation to record edges adjacent to owned vertices (0 and 1). The index array `idx` records each vertex's edge distribution: for vertex i, `idx[i]` and `idx[i+1]` indicate the beginning and ending offsets of its outgoing/incoming edges to this particular partition. The array `nbr` records the neighbors of these edges (sources for incoming edges or destinations for outgoing ones).

Yet, from our experiments and performance analysis, we find that the basic CSR/CSC format is insufficient. More specifically, the index array `idx` can become a scaling bottleneck, as its size remains at $O(|V|)$ while the size of edge storage is reduced proportionally at $O(|E|/p)$ as p grows. For example, in Figure 6, the partition has only 4 dense mode edges, but has to traverse the 7-element ($|V|+1$) `idx` array, making the processing of adjacent vertices (rather than edges) the bottleneck in dense mode computation. A conventional solution to this is to compress the vertex ID space. This comes at the cost of converting IDs between global and local states, which adds other non-negligible overhead to the system.

To resolve the bottleneck in a lightweight fashion, we use two schemes for enhancing the index array in the two modes, as described below and illustrated in Figure 6:

- *Bitmap Assisted Compressed Sparse Row*: for sparse mode edges, we add an existence bitmap `ext`, which marks whether each vertex has outgoing edges in this partition. For example, only vertex 0, 2, and 4 are present, indicated by the bitmap 101010.
- *Doubly Compressed Sparse Column*: for dense mode edges, we use a doubly-compression scheme [9] to store only vertices with incoming edges (`vtx`) and their corresponding edge offsets (`off`, where (`off[i+1]-off[i]`) indicates the number of local incoming edges vertex `vtx[i]` has). For example, only vertex 1, 2, 3, and 5 has local incoming edges.

Both schemes reduce memory accesses required in edge processing. In the dense mode, where all the vertices in a local partition has to be processed, the compressed indices enable Gemini to only access $O(|V_i'|)$ vertex indices reduced from $O(|V|)$. In the sparse mode, the bitmap eliminates the lookups into `idx` of vertices that do not have outgoing edges in the local partition, which occurs frequently when the graph is partitioned.

4.3 Locality-Aware Chunking

We now discuss how Gemini actually decides where to make the $p-1$ cuts when creating p contiguous vertex chunks, using a locality-aware criterion.

Traditionally, graph partitioning pursues *even distribution* of either the vertices (in vertex-centric scenarios) or the edges (in edge-centric scenarios) to enhance load balance.

While Gemini's chunk-based partitioning is vertex-centric, we find that balanced vertex distribution exhibits poor load balance due to the power-law distribution [15] of vertex degrees exhibited in most real-world graphs, often considered a disadvantage of vertex-centric solutions compared with their edge-centric counterparts [16].

However, with chunk-based partitioning, even balanced edge chunking, with $|E|/p$ edges per partition uniformly brings significant load imbalance. Our closer examination finds that vertex access locality (one of the performance focal points of Gemini's chunk-based partitioning) differs significantly across partitions despite balanced edge counts, incurring large variation in $|V_i|$, the size of vertex chunks.

While dynamic load balancing techniques [26, 41] such as workload re-distribution might help, they incur heavy costs and extra complexities that go against Gemini's lightweight design. Instead, Gemini employs a locality-aware enhancement, adopting a hybrid metric that considers both owned vertices and dense mode edges in setting the balancing criteria. More specifically, the vertex array V is split in a manner so that each partition has a balanced value of $\alpha \cdot |V_i| + |E_i^D|$. Here α is a configurable parameter, set empirically to $8 \cdot (p-1)$ as Gemini's default configuration in our experiments, which might be adjusted according to hardware configurations or application/input properties.

The intuition behind such hybrid metric is that one needs to take into account the computation complexity from both the vertex and the edge side. Here the size of the partition, in terms of $|V_i|$ and $|E_i^D|$, not only affects the amount of work ($|E_i^D|$), but also the memory access locality ($|V_i|$). To analyze the joint implication of specific system, algorithm, and input features on load balancing and enable adaptive chunk partitioning (*e.g.*, automatic configuration of α) is among our ongoing investigations.

4.4 NUMA-Aware Sub-Partitioning

An interesting situation with today's high-performance cluster is that the scale of intra-node parallelisms could easily match or exceed that of inter-node levels. For instance, our testbed has 8 nodes, each with 24 cores. Effectively exploiting both intra- and inter-node hardware parallelism is crucial to the overall performance of distributed graph processing.

Most modern servers are built on the NUMA (Non-Uniform Memory Access) architecture, where memory is physically distributed on multiple sockets, each typically containing a multi-core processor with local memory. Sockets are connected through high-speed interconnects into a global cache-coherent shared-memory system. Access to local memory is faster than to remote memory (attached to other sockets), both in terms of lower latencies and higher bandwidths [14], making it appealing to minimize *inter-socket* accesses.

Gemini's chunk-based graph partitioning demonstrates another advantage here, by allowing the system to recursively apply sub-partitioning in a consistent manner, potentially with different optimizations applicable at each particular level. Within a node, Gemini applies NUMA-aware sub-partitioning across multiple sockets: for each node containing s sockets, the vertex chunk V_i is further cut into *s sub-chunks*, one for each socket. Edges are assigned to corresponding sockets, using the same rules as in inter-node partitioning (Section 4.1).

NUMA-aware sub-partitioning boosts the performance on NUMA machines significantly. It retains the natural locality present in input vertex arrays, as well as lightweight partitioning and bookkeeping. With smaller yet densely processed sub-chunks, both sequential accesses to edges and random accesses to vertices are likely to fall into the local memory, facilitating faster memory access and higher LLC (last level cache) utilization simultaneously.

5 Task Scheduling

Like most recent distributed graph processing systems [3, 11, 12, 16, 17, 23, 26, 32, 41, 43], Gemini follows the Bulk Synchronous Parallel (BSP) model [53]. In each iteration of edge processing, Gemini co-schedules computation and communication tasks in a cyclic ring order to effectively overlap inter-node communication with computation. Within a node, Gemini employs a fine-grained work-stealing scheduler with shared pre-computed chunk counters to enable dynamic load balancing at a fine granularity. Below we discuss these two techniques in more detail.

5.1 Co-Scheduling of Computation and Communication Tasks

Inspired by the well-optimized implementation of collective operations in HPC communication libraries, such as `AllGather` in MPI, Gemini organizes cluster nodes in a ring, with which message sending and receiving operations are coordinated in a balanced cyclic manner, to reduce network congestion and maximize aggregate message passing throughput. Such orchestrated communication tasks are further carefully overlapped with computation tasks, to hide network communication costs.

On a cluster node with c cores, Gemini maintains an OpenMP pool of c threads for parallel vertex-centric edge processing, performing the `signal` and `slot` tasks. Each thread is bound to specific sockets to work with NUMA-aware sub-partitioning. In addition, two helper

threads per node are created for inter-node message sending/receiving operations via MPI.

Here again thanks to Gemini's chunk-based partitioning and CSR/CSC organization of edges, it can naturally batch messages destined to the same partition in both sparse and dense modes for high-performance communication. Moreover, the batched messages enable us to schedule the tasks in a simple partition-oriented fashion. Figure 7 illustrates the co-scheduled ordering of the four types of tasks, using the dense mode in the first partition ($node_0$) of the previous figure as an example.

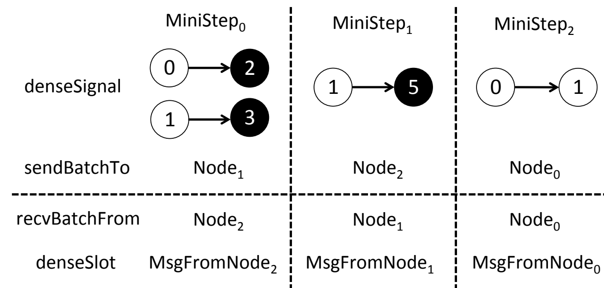

Figure 7: Example of co-scheduled computation and communication tasks on $node_0$

The iteration is divided into p mini-steps, during each of which $node_i$ communicate with one peer node, starting from $node_{i+1}$ back to itself. In the particular example shown in Figure 7, there are three such stages, where $node_0$ communicates with $node_1$, $node_2$, and $node_0$ respectively. In each mini-step, the node goes through local `denseSignal` processing, message send/receive, and final local `denseSlot` processing. For example, here in the first mini-step, local mirrors of vertices 2 and 3 (owned by $node_1$) pull updates from vertices 0 and 1 (owned by self), creating a batched message ready for $node_1$, after whose transmission $node_0$ expects a similar batched message from $node_2$ and processes that in `denseSlot`. In the next mini-step, similar update is pulled by all local mirrors owned by $node_2$ (only vertex 5 in this figure), followed by communication and local processing. The process goes on until $node_0$ finally "communicates" with itself, where it simply pulls from locally owned neighbors (vertex 1 from 0). As separate threads are created to execute the CPU-light message passing tasks, computation is effectively overlapped with communication.

5.2 Fine-Grained Work-Stealing

While inter-node load balance is largely ensured through the locality-aware chunk-based partitioning in Gemini, the hybrid vertex-edge balancing gets more and more challenging when the partition goes smaller, from nodes to sockets, then to cores. With smaller partitions, there are fewer flexibilities for tuning the α parameter to achieve inter-core load balance, especially for graphs with high per-vertex degree variances.

Leveraging shared memory not available to inter-node load balancing, Gemini employs a fine-grained work-stealing scheduler for intra-node edge processing. While the per-socket edge processing work is preliminarily partitioned with a locality-aware balanced manner across all the cores as a starting point, each thread only grabs a small *mini-chunk* of vertices to process (`signal/slot`) during the OpenMP parallel region. Again, due to our chunk-based partitioning scheme, this refinement retains contiguous processing, and promotes efficient cache utilization and message batching. Bookkeeping is also easy, as it only requires one counter per core to mark the current mini-chunk's starting offset, shared across threads and accessed through atomic operations. The default Gemini setting of mini-chunk size is 64 vertices, as used in our experiments.

Each thread first tries to finish its own per-core partition, then starts to steal mini-chunks from other threads' partitions. Compared with finely interleaved mini-chunk distribution from the beginning, this enhances memory access by taking advantage of cache prefetching. Also, this delays contention involved in atomic additions on the shared per-core counters to the epilogue of the whole computation. At that point, the cost is clearly offset by improved inter-core load balance.

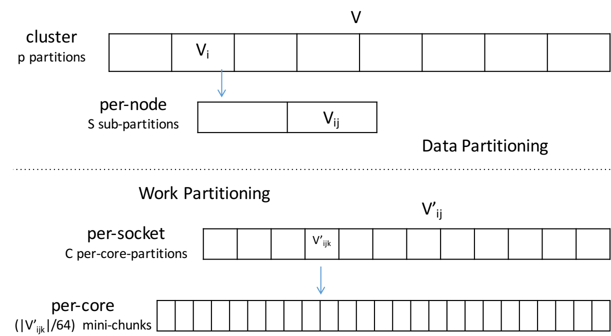

Figure 8: Hierarchical view of Gemini's chunking

Finally, we summarize Gemini's multi-level chunk-based partitioning in Figure 8, all the way from node-level, to socket-level, core-level, and finally to the mini-chunk granularity for inter-core work stealing. The illustration depicts the partitioning scenario in our actual test cluster with 8 nodes, 2 sockets per node, 12 cores per socket, and 64 vertices per mini-chunk. As shown here, such simple chunk-based partitioning can be refined in a hierarchical way, retaining access locality in edge processing continuously.

6 Implementation

Gemini is implemented in around 2,800 lines of C++ code, using MPI for inter-process communication and `libnuma` for NUMA-aware memory allocation. Below we discuss selected implementation details.

Graph Loading: When Gemini loads a graph from input file, each node reads its assigned contiguous portion in parallel. Edges are loaded sequentially into an edge buffer in batches, where they undergo an initial pass. Compared to common practice in existing systems, this reduces the memory consumption of the loading phase significantly, making it possible to load graphs whose scales approach the aggregate memory capacity of the whole cluster. For symmetric graphs, Gemini only stores the graph topology data of one mode, as sparse mode edges in this case are equivalent to the dense mode ones.

Graph Partitioning: When loading edges, each node calculates the local degree of each vertex. Next, an `AllReduce` operation collects such degree information for chunking the vertex set as discussed in Section 4. Each node can then determine the cuts locally without communication. Edges are then re-loaded from file and distributed to target nodes accordingly for constructing local subgraphs.

Memory Allocation: All nodes share the the node-level partitioning boundaries for inter-node message passing, while the socket-level sub-partition information is kept node-private. Each node allocates entire vertex arrays in shared memory. However, Gemini only touches data within its own vertex chunk, splitting the per-node vertex partition and placing the sub-chunks on corresponding sockets. The sub-partitioned graph topology datasets, namely edges and vertex indices, also adopts NUMA-aware allocation to promote localized memory accesses.

Mode Selection: Gemini follows Ligra's mode switching mechanism. For each `ProcessEdges` operation, Gemini first invokes an internal operation (defined via its `ProcessVertices` interface) to get the number of active edges, then determines the mode to use for the coming interation of processing.

Parallel Processing: When the program initializes, Gemini pins each OpenMP thread to specific sockets to prevent thread migration. For work-stealing, each thread maintains its status (`WORKING` or `STEALING`), current mini-chunk's start offset, and the pre-computed end offset, which are accessible to other threads and allocated in a NUMA-aware aligned manner to avoid false-sharing and unnecessary remote memory accesses (which should only happen in stealing stages). Each thread starts working from its own partition, changes the status when finished, and tries to steal work from threads with higher ranks in a cyclic manner. Concurrency control is via OpenMP's implicit synchronization mechanisms.

Message Passing: Gemini runs one process on each node, using MPI for inter-node message passing. At the inter-socket level, each socket produces/consumes messages through per-socket send and receive buffers in shared memory to avoid extra memory copies and perform NUMA-aware message batching.

7 Evaluation

We evaluate Gemini on the 8-node cluster, whose specifications are given in Section 2, running CentOS 7.2.1511. Intel ICPC 16.0.1 is used for compilation.

The graph datasets used for evaluation are shown in Table 2. Our evaluation uses five representative graph analytics applications: PageRank (*PR*), connected components (*CC*[1]), single source shortest paths (*SSSP*[2]), breadth first search (*BFS*), and betweenness centrality (*BC*). For comparison, we also evaluated state-of-the-art distributed graph processing systems, including PowerGraph (v2.2), GraphX (v2.0.0), and PowerLyra (v1.0), as well as shared-memory Ligra (20160826) and Galois (v2.2.1). For each system, we make our best effort to optimize the performance on every graph by carefully tuning the parameters, such as the partitioning method, the number of partitions, JVM options (for GraphX), the used algorithm (for Galois), etc. To get stable performance, we run *PR* for 20 iterations, and run *CC*, *SSSP*, *BFS*, and *BC* till convergence. The execution time is reported as elapsed time for executing the above graph algorithms (average of 5 runs) and does not include loading or partitioning time.

Graph	$\lvert V \rvert$	$\lvert E \rvert$
enwiki-2013	4,206,785	101,355,853
twitter-2010	41,652,230	1,468,365,182
uk-2007-05	105,896,555	3,738,733,648
weibo-2013	72,393,453	6,431,150,494
clueweb-12	978,408,098	42,574,107,469

Table 2: Graph datasets [5, 6, 7, 20] used in evaluation.

7.1 Overall Performance

As Gemini aims to provide scalability on top of efficiency, to understand the introduced overhead, we first take a zoom-in view of its single-node performance, using the five applications running on the *twitter-2010* graph. Here, instead of using distributed graph-parallel systems, we compare Gemini with two state-of-the-art shared-memory systems, Ligra and Galois, which we have verified to have superior performance compared

[1] Gemini makes the input graphs undirected when computing *CC*.
[2] A random weight between 0 and 100 is assigned to each edge.

with single-node executions of all the aforementioned distributed systems.

Application	Ligra	Galois	Gemini
PR	21.2	19.3	**12.7**
CC	6.51	**3.59***	4.93
SSSP	**2.81**	3.33	3.29
BFS	**0.347**	0.528	0.468
BC	2.45	3.94*	**1.88**

Table 3: 1-node runtime (in seconds) on input graph *twitter-2010*. The best times are marked in bold. "*" indicates where different algorithm is adopted (i.e. union-find for *CC* and asynchronous for *BC*).

Table 3 presents the performance of evaluated systems. Though with communication complexity designed for distributed execution, Gemini outperforms Ligra and Galois for *PR* and *BC*, and ranks the second for *CC*, *SSSP*, and *BFS*. With the use of NUMA-aware sub-partitioning, Gemini benefits from faster memory access and higher LLC utilization in edge processing, thanks to significantly reduced visits to remote memory. Unlike the NUMA-oblivious access patterns of Ligra and Galois, Gemini's threads only visit remote memory for work-stealing and message-passing.

Meanwhile, the distributed design inevitably brings additional overhead. The messaging abstraction (*i.e.*, batched messages produced by signals and consumed by slots) introduces extra memory accesses. This creates a major performance constraint for less computation-intensive applications like *BFS*, where the algorithm does little computation while the numbers of both visited edges and generated messages are in the order of $O(\Sigma|V_i|)$.[3] In contrast, most other applications access *all* adjacent edges of each active vertex, creating edge processing cost proportional to the number of active edges and sufficient to mask the message generation overhead.

Also, vertex state propagation in shared-memory systems employs direct access to the *latest* vertex states, while Gemini's BSP-based communication mechanism can only fetch the neighboring states through message passing in a super-step granularity. Therefore, its vertex state propagation lags behind that of shared-memory systems, forcing Gemini to run more iterations than Ligra and Galois for label-propagation-style applications like *CC* and *SSSP*.

Overall, with a relatively low cost paid to support distributed execution, Gemini can process much larger graphs by scaling out to more nodes and to work quite efficiently on single-node multi-core machines, allowing it to handle diverse application-platform combinations.

[3]In BFS's dense mode, edge processing at a vertex completes as soon as it successfully "pulls" from any of its neighbors [4].

Graph	PowerG.	GraphX	PowerL.	Gemini	Speedup (×times)
PR					
enwiki-2013	9.05	30.4	7.27	0.484	15.0
twitter-2010	40.3	216	26.9	3.02	8.91
uk-2007-05	64.9	416	58.9	1.48	39.8
weibo-2013	117	-	100	8.86	11.3
clueweb-12	-	-	-	31.1	n/a
CC					
enwiki-2013	4.61	16.5	5.02	0.237	19.5
twitter-2010	29.1	104	22.0	1.22	18.0
uk-2007-05	72.1	-	63.4	1.76	36.0
weibo-2013	56.5	-	58.6	2.62	21.6
clueweb-12	-	-	-	25.7	n/a
SSSP					
enwiki-2013	16.5	151	17.1	0.514	32.1
twitter-2010	12.5	108	10.8	1.15	9.39
uk-2007-05	117	-	143	3.45	33.9
weibo-2013	63.2	-	60.6	4.24	14.3
clueweb-12	-	-	-	56.9	n/a
GEOMEAN					19.1

Table 4: 8-node runtime (in seconds) and improvement of Gemini over the best of other systems. "-" indicates failed execution.

Table 4 reports the 8-node performance of PowerGraph, GraphX, PowerLyra, and Gemini, running *PR*, *CC*, and *SSSP* on all the tested graphs (*BFS* and *BC* results are omitted as their implementations are absent in other evaluated systems). The results show that Gemini outperforms the fastest of other systems in all cases significantly (19.1× on average), with up to 39.8× for *PR* on the *uk-2007-05* graph. For the *clueweb-12* graph with more than 42 billion edges, Gemini is able to complete *PR*, *CC*, and *SSSP* in 31.1, 25.7, and 56.9 seconds respectively on the 8-node cluster while *all* other systems fail to finish due to excessive memory consumption.

Graph	Raw	PowerGraph	Gemini
enwiki-2013	0.755	13.1	4.02
twitter-2010	10.9	138	32.1
uk-2007-05	27.8	322	73.1
weibo-2013	47.9	561	97.5
clueweb-12	318	-	597

Table 5: Peak 8-node memory consumption (in GB). "-" indicates incompletion due to running out of memory.

The performance gain mostly comes from the largely reduced distributed overhead. Table 5 compares the memory consumption of PowerGraph and Gemini. The raw graph size (with each edge in two 32-bit integers) is also presented for reference. PowerGraph needs memory more than 10× the raw size of a graph to process it. The larger memory footprint brings more instructions

and memory accesses, and lowers the cache efficiency.

In contrast, while Gemini needs to store two copies of edges (in CSR and CSC respectively) due to its dual-mode propagation, the actual memory required is well controlled. Especially, the relative space overhead decreases for larger graphs (*e.g.*, within 2× of the raw size for *clueweb-12*). Gemini's abstraction (chunk-based partitioning scheme, plus the sparse-dense signal-slot processing model) adds very little overhead to the overall system and preserves (or enhances when more nodes are used) access locality present in the original graph. The co-scheduling mechanism hides the communication cost effectively under the high-speed Infiniband network. Locality-aware chunking and fine-grained work-stealing further improves inter-node and intra-node load balance. These optimizations together enable Gemini to provide scalability on top of efficiency.

7.2 Scalability

Next, we examine the scalability of Gemini, starting from intra-node evaluation using 1 to 24 cores to run *PR* on the *twitter-2010* graph (Figure 9). Overall the scalability is quite decent, achieving speedup of 1.9, 3.7, and 6.8 at 2, 4, and 8 cores, respectively. As expected, as more cores are used, inter-core load balancing becomes more challenging, synchronization cost becomes more visible, and memory bandwidth/LLC contention becomes intensified. Still, Gemini is able to achieve a speedup of 9.4 at 12 cores and 15.5 at 24 cores.

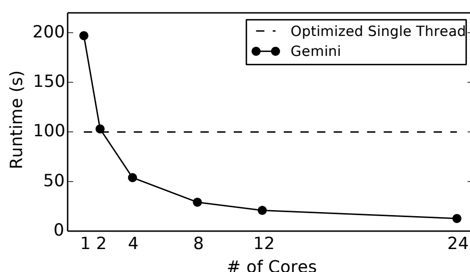

Figure 9: Intra-node scalability (*PR* on *twitter-2010*)

To further evaluate Gemini's computation efficiency, we compare it with the optimized single-thread implementation (which sorts edges in a Hilbert curve order [33]), shown as the dashed horizontal line in Figure 9. Using the COST metric (i.e. how many cores a parallel/distributed solution needs to outperform the optimized single-thread implementation), Gemini's number is 3, which is lower than those of other systems measured [33], though Gemini's 2-core execution time is only 3.1% higher than the optimized single-thread implementation. Considering Gemini's distributed nature, a COST close to 2 illustrates its optimized computation efficiency and lightweight distributed execution overhead.

Figure 10 shows the inter-node scalability results, comparing Gemini with PowerLyra, which we found to have the best performance and scalability for our test cases among existing open-source systems. Due to its higher memory consumption, PowerLyra is not able to complete in several test cases, as indicated by the missing data points. All results are normalized to Gemini's best execution time of the test case in question. It shows that though focused on computation optimization, Gemini is able to deliver inter-node scalability very similar to that by PowerLyra, approaching linear speedup with large graphs (*weibo-2013*). With the smallest graph (*enwiki-2013*), as expected, the scalability is poor for both systems as communication time dominates the execution.

For *twitter-2010*, Gemini has poor scaling after 4 nodes, mainly due to the emerging bottleneck from vertex indices access and message production/consumption. This is confirmed by the change of subgraph dimensions shown in Table 6: when more nodes are used, both $|E_i|$ and $|V_i|$ scales down perfectly, reducing edge processing cost. The vertex set including mirrors, V_i', however, does not shrink accordingly, making its processing cost increasingly significant.

$p \cdot s$	T_{PR} (s)	$\Sigma\|V_i\|/(p\cdot s)$	$\Sigma\|E_i\|/(p\cdot s)$	$\Sigma\|V_i'\|/(p\cdot s)$
$1\cdot 2$	12.7	20.8M	734M	27.6M
$2\cdot 2$	7.01	10.4M	367M	19.6M
$4\cdot 2$	3.88	5.21M	184M	13.5M
$8\cdot 2$	3.02	2.60M	91.8M	10.5M

Table 6: Subgraph sizes with growing cluster size

7.3 Design Choices

Below we evaluate the performance impact of several major design choices in Gemini. Though it is tempting to find out the relative significance among these optimizations themselves, we have found it hard to compare the contribution of individual techniques, as they often assist each other (such as chunk-based partitioning and intra-node work-stealing). In addition, when we incrementally add these optimizations to a baseline system, the apparent gains measured highly depend on the order used in such compounding. Therefore we present and discuss the advantages of individual design decisions, where results do not indicate their relative strength.

7.3.1 Adaptive Sparse-Dense Dual Mode

Adaptive switching between sparse and dense modes according to the density of active edges improves the performance of Gemini significantly. We propose an exper-

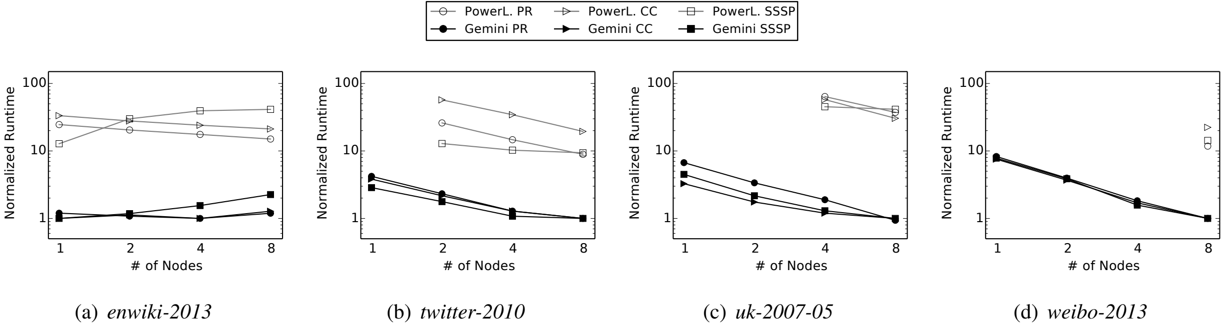

Figure 10: Inter-node scalability of PowerLyra and Gemini

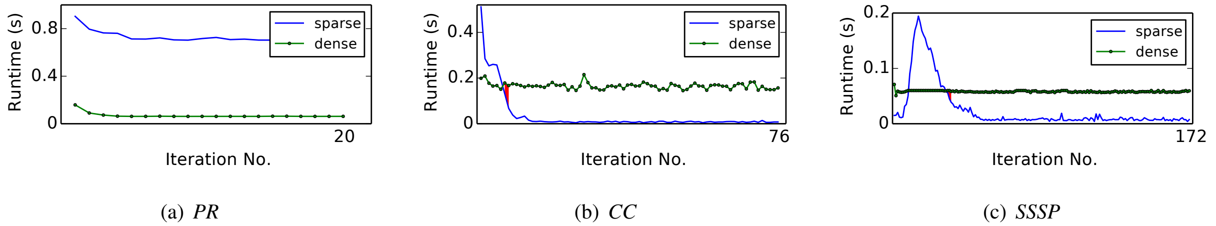

Figure 11: Gemini's per-iteration runtime in sparse and dense modes (*uk-2007-05*). Red regions indicate iterations where Gemini's adaptive engine chooses sub-optimal modes.

iment by forcing Gemini to run under the two modes for each iteration respectively to illustrate the necessities of the dual mode abstraction.

As shown in Figure 11, the performance gap between sparse and dense modes is quite significant, for all three applications. For PR, the dense mode consistently outperforms the sparse one. For CC, the dense mode performs better at the first few iterations when most of the vertices remain active, while the sparse mode is more effective when more vertices reach convergence. For SSSP, the sparse mode outperforms the dense mode in most iterations, except in a stretch of iterations where many vertices get updated. Gemini is able to adopt the better mode in most iterations, except 2 out of 76 for CC and 5 out of 172 iterations for SSSP. These "mis-predictions" are slightly sub-optimal as they happen, as expected, around the intersection of the two modes' performance curves.

7.3.2 Chunk-Based Partitioning

Next, we examine the effectiveness of Gemini's chunk-based partitioning through an experiment comparing it against hash-based partitioning[4].

Figure 12 exhibits the performance of Gemini using these two partitioning methods on *twitter-2010* and *uk-2007-05*, sampled to represent social and web graphs, respectively. Gemini's chunk-based partitioning outperforms the hash-based solution for both graphs. The performance improvement is especially significant for the web graph *uk-2007-05*, with more than 5.44× speedup. The reason behind is the locality-preserving property of chunk-based partitioning. Hash-based partitioning, in contrast, loses the natural locality in the original graph. As a result, hash-based partitioning produces not only higher LLC miss rates, but also a large number of mirrors in each partition, higher communication costs, and more

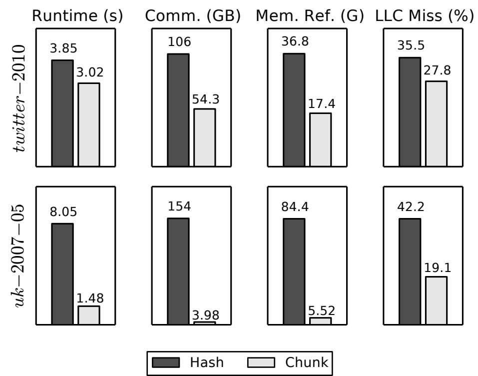

Figure 12: Hash- vs. chunk-based partitioning (PR on *twitter-2010* and *uk-2007-05*)

[4] We integrate the hash-based scheme (assigning vertex x to partition $x\%p$) into Gemini by re-ordering vertices according to the hashing result before chunking them.

memory references, for master-mirror message passing and vertex index accesses.

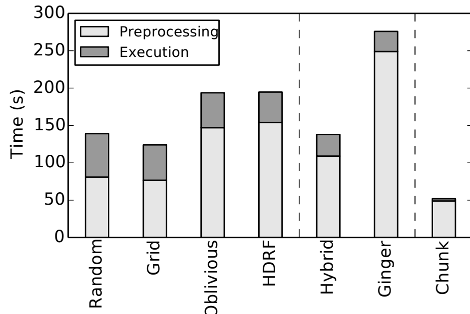

Figure 13: Preprocessing/execution time (*PR* on *twitter-2010*) with different partitioning schemes

Figure 13 shows the preprocessing time (loading plus partitioning) of different partitioning methods [12, 16, 24, 39] used by PowerGraph, PowerLyra, and Gemini on *twitter-2010*, with PR execution time given as reference. While it appears that preprocessing takes much longer than the algorithm execution itself, such preprocessing only poses a one-time cost, while the partitioned graph data can be re-used repeatedly by different applications or with different parameters.

NUMA-aware sub-partitioning plays another important role, as demonstrated by Figure 14 comparing sample Gemini performance with and without it. Without socket-level sub-partitioning, interleaved memory allocation leaves all accesses to the graph topology, vertex states, and message buffers distributed across both sockets. With socket-level sub-partitioning applied, instead, remote memory accesses are significantly trimmed, as they only happen when stealing work from or accessing messages produced by other sockets. The LLC miss rate and average memory access latency also decrease thanks to having per-socket vertex chunks.

7.3.3 Enhanced Vertex Index Representation

Table 7 presents the improvement brought by using bitmap assisted compressed sparse row and doubly compressed sparse column, with three applications on two input graphs. Compared with the original CSR/CSC formats, these enhanced data structures reduces memory consumption by 19-24%. They also eliminate many unnecessary memory accesses, bringing additional performance gain.

7.3.4 Load Balancing

Next, Table 8 portraits the benefit of Gemini's locality-aware chunking, by giving the number of owned vertices and dense mode edges in the *most time-consuming* partition. Compared with alternatives that aim at balancing vertex or edge counts, Gemini improves load balance by considering both vertex access locality and number of edges to be processed.

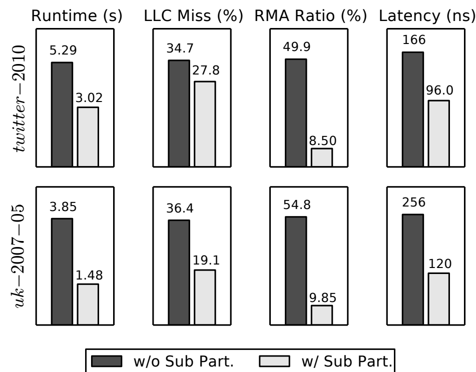

Figure 14: Impact of socket-level sub-partitioning (*PR* on *twitter-2010* and *uk-2007-05*)

Graph		*twitter-2010*	*uk-2007-05*
Mem. Reduction		19.4%	24.3%
Speedup	*PR*	1.25	2.76
	CC	1.11	1.30
	SSSP	1.14	1.98

Table 7: Impact of enhanced vertex index representation

Balanced By	Runtime (s)	$\lvert V_i \rvert$	$\lvert E_i^D \rvert$
$\lvert V_i \rvert$	5.51	5.21M	957M
$\lvert E_i^D \rvert$	3.95	18.1M	183M
$\alpha \cdot \lvert V_i \rvert + \lvert E_i^D \rvert$	3.02	0.926M	423M

Table 8: Impact of locality-aware chunking (*PR* on *twitter-2010*)

Finally, we evaluate the effect of Gemini's fine-grained work-stealing by measuring the improvement by three intra-node load balancing strategies. More specifically, we report the relative speedup of (1) static, pre-balanced per-core work partitions using our locality-aware chunking, (2) work-oblivious stealing, and (3) the integration of both (as adopted in Gemini), over the baseline using static scheduling. Table 9 lists the results. As expected, static core-level work partitioning is not enough to ensure effective multi-core utilization. Yet, pre-computed per-core work partitions do provide a good starting point when working jointly with work stealing.

Strategy	*twitter-2010*	*uk-2007-05*
Balanced partition	1.25	1.66
Stealing	1.55	1.93
Balanced partition + stealing	1.66	2.18

Table 9: *PR* speedup (over static scheduling) with different intra-node load balancing strategies

8 Related Work

We have discussed and evaluated several most closely related graph-parallel systems earlier in the paper. Here we give a brief summary of related categories of prior work.

A large number of graph-parallel systems [3, 10, 11, 12, 16, 17, 21, 22, 23, 26, 29, 30, 32, 36, 41, 42, 43, 44, 47, 49, 55, 56, 57, 59, 60] have been proposed for efficient processing of graphs with increasing scales. Gemini is inspired by prior systems in various aspects, but differs from them by taking a holistic view on system design toward single-node efficiency and multi-node scalability.

Push vs. Pull: Existing distributed graph processing systems either adopt a push-style [3, 26, 32, 43, 44] or a pull-style [11, 12, 16, 17, 23, 30] model, or provide both while used separately [13, 19, 22]. Recognizing the importance of a model that adaptively combines push and pull operators as shown by shared-memory approaches [4, 36, 47, 57], Gemini extends the hybrid push-pull model from shared-memory to distributed-memory settings through a signal-slot abstraction to decouple communication from computation, which is novel in the context of distributed graph processing.

Data Distribution: Traditional literature in graph partitioning [8, 12, 16, 24, 25, 30, 32, 39, 48] puts the main focus on reducing communication cost and load imbalance, without enough attention on the introduced overhead to distributed graph processing. Inspired by the implementation of several single-node graph processing systems [29, 42, 49, 57, 60], Gemini adopts a chunk-based partitioning scheme that enables a low-overhead scaling out design. When applying the chunking method in a distributed fashion, we address new challenges, including the sparsity in vertex indices, inter-node load imbalance, and intra-node NUMA issues, with further optimizations to accelerate computation.

Communication and Coordination: GraM [55] designs an efficient RDMA-based communication stack to overlap communication and computation for scalability. Gemini achieves similar goals by co-scheduling computation and communication tasks in a partition-oriented ring order, which is inspired by the implementation of collective operations in MPI [51], and can work effectively without the help of RDMA. PGX.D [22] highlights the importance of intra-node load balance to performance and proposes an edge chunking method. Gemini extends the idea by integrating chunk-based core-level work partitioning into a fine-grained work-stealing scheduler, which allows it to achieve better multi-core utilization.

There also exist many systems that focus on query processing [40, 46, 54], temporal analytics [13, 19, 27, 31], machine learning and data mining [50, 58], or more general tasks [34, 35, 45] on large-scale graphs. It would be interesting to explore how Gemini's computation-centric design could be applied to these systems.

9 Conclusion

In this work, we investigated computation-centric distributed graph processing, re-designing critical system components such as graph partitioning, graph representation and update propagation, task/message scheduling, and multi-level load balancing surrounding the theme of improving computation efficiency on modern multi-core cluster nodes. Our development and evaluation reveal that (1) effective system resource utilization relies on building low-overhead distributed designs upon optimized single-node computation efficiency, and (2) low-cost chunk-based partitioning preserving data locality across multiple levels of parallelism performs surprisingly well, and opens up many opportunities for subsequent optimizations throughout the system.

Meanwhile, through the evaluation of Gemini and other open-source graph processing systems, we have noticed that performance, scalability, and the location of bottleneck are highly dependent on the complex interaction between algorithms, input graphs, and underlying systems. Relative performance results comparing multiple alternative systems reported in papers (including this one) sometimes cannot be replicated with different platform configurations or input graphs. This also highlights the need of adaptive systems that customizes its decisions based on dynamic application, data, and platform behaviors.

Acknowledgments

We sincerely thank all the reviewers for their insightful comments and suggestions. We also thank Haibo Chen, Rong Chen, and Tej Chajed for their valuable feedback during our preparation of the final paper. This work is supported in part by the National Grand Fundamental Research 973 Program of China under Grant No. 2014CB340402, and the National Science Fund for Distinguished Young Scholars under Grant No. 61525202.

References

[1] https://en.wikipedia.org/wiki/Signals_and_slots.

[2] APOSTOLICO, A., AND DROVANDI, G. Graph compression by bfs. *Algorithms 2*, 3 (2009), 1031–1044.

[3] AVERY, C. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara* (2011).

[4] BEAMER, S., ASANOVIĆ, K., AND PATTERSON, D. Direction-optimizing breadth-first search. *Scientific Programming 21*, 3-4 (2013), 137–148.

[5] BOLDI, P., ROSA, M., SANTINI, M., AND VIGNA, S. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World wide web* (2011), ACM, pp. 587–596.

[6] BOLDI, P., SANTINI, M., AND VIGNA, S. A large time-aware graph. *SIGIR Forum 42*, 2 (2008), 33–38.

[7] BOLDI, P., AND VIGNA, S. The webgraph framework i: compression techniques. In *Proceedings of the 13th international conference on World Wide Web* (2004), ACM, pp. 595–602.

[8] BOURSE, F., LELARGE, M., AND VOJNOVIC, M. Balanced graph edge partition. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining* (2014), ACM, pp. 1456–1465.

[9] BULUÇ, A., AND GILBERT, J. R. On the representation and multiplication of hypersparse matrices. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on* (2008), IEEE, pp. 1–11.

[10] BULUÇ, A., AND GILBERT, J. R. The combinatorial blas: Design, implementation, and applications. *International Journal of High Performance Computing Applications* (2011), 1094342011403516.

[11] CHEN, R., DING, X., WANG, P., CHEN, H., ZANG, B., AND GUAN, H. Computation and communication efficient graph processing with distributed immutable view. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing* (2014), ACM, pp. 215–226.

[12] CHEN, R., SHI, J., CHEN, Y., AND CHEN, H. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM, p. 1.

[13] CHENG, R., HONG, J., KYROLA, A., MIAO, Y., WENG, X., WU, M., YANG, F., ZHOU, L., ZHAO, F., AND CHEN, E. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM european conference on Computer Systems* (2012), ACM, pp. 85–98.

[14] DAVID, T., GUERRAOUI, R., AND TRIGONAKIS, V. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 33–48.

[15] FALOUTSOS, M., FALOUTSOS, P., AND FALOUTSOS, C. On power-law relationships of the internet topology. In *ACM SIGCOMM Computer Communication Review* (1999), vol. 29, ACM, pp. 251–262.

[16] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI* (2012), vol. 12, p. 2.

[17] GONZALEZ, J. E., XIN, R. S., DAVE, A., CRANKSHAW, D., FRANKLIN, M. J., AND STOICA, I. Graphx: Graph processing in a distributed dataflow framework.

[18] GREGOR, D., AND LUMSDAINE, A. The parallel bgl: A generic library for distributed graph computations.

[19] HAN, W., MIAO, Y., LI, K., WU, M., YANG, F., ZHOU, L., PRABHAKARAN, V., CHEN, W., AND CHEN, E. Chronos: a graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 1.

[20] HAN, W., ZHU, X., ZHU, Z., CHEN, W., ZHENG, W., AND LU, J. Weibo, and a tale of two worlds. In *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2015* (2015), ACM, pp. 121–128.

[21] HAN, W.-S., LEE, S., PARK, K., LEE, J.-H., KIM, M.-S., KIM, J., AND YU, H. Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining* (2013), ACM, pp. 77–85.

[22] HONG, S., DEPNER, S., MANHARDT, T., VAN DER LUGT, J., VERSTRAATEN, M., AND CHAFI, H. Pgx.d: A fast distributed graph processing engine. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2015), SC '15, ACM, pp. 58:1–58:12.

[23] HOQUE, I., AND GUPTA, I. Lfgraph: Simple and fast distributed graph analytics. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems* (2013), ACM, p. 9.

[24] JAIN, N., LIAO, G., AND WILLKE, T. L. Graphbuilder: scalable graph etl framework. In *First International Workshop on Graph Data Management Experiences and Systems* (2013), ACM, p. 4.

[25] KARYPIS, G., AND KUMAR, V. Parallel multilevel series k-way partitioning scheme for irregular graphs. *Siam Review 41*, 2 (1999), 278–300.

[26] KHAYYAT, Z., AWARA, K., ALONAZI, A., JAMJOOM, H., WILLIAMS, D., AND KALNIS, P. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), ACM, pp. 169–182.

[27] KHURANA, U., AND DESHPANDE, A. Efficient snapshot retrieval over historical graph data. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on* (2013), IEEE, pp. 997–1008.

[28] KWAK, H., LEE, C., PARK, H., AND MOON, S. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web* (New York, NY, USA, 2010), ACM, pp. 591–600.

[29] KYROLA, A., BLELLOCH, G. E., AND GUESTRIN, C. Graphchi: Large-scale graph computation on just a pc. In *OSDI* (2012), vol. 12, pp. 31–46.

[30] LOW, Y., BICKSON, D., GONZALEZ, J., GUESTRIN, C., KYROLA, A., AND HELLERSTEIN, J. M. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment 5*, 8 (2012), 716–727.

[31] MACKO, P., MARATHE, V. J., MARGO, D. W., AND SELTZER, M. I. Llama: Efficient graph analytics using large multiversioned arrays. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on* (2015), IEEE, pp. 363–374.

[32] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (2010), ACM, pp. 135–146.

[33] MCSHERRY, F., ISARD, M., AND MURRAY, D. G. Scalability! but at what cost? In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2015), HOTOS'15, USENIX Association, pp. 14–14.

[34] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 439–455.

[35] NELSON, J., HOLT, B., MYERS, B., BRIGGS, P., CEZE, L., KAHAN, S., AND OSKIN, M. Latency-tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (2015), pp. 291–305.

[36] NGUYEN, D., LENHARTH, A., AND PINGALI, K. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 456–471.

[37] OUSTERHOUT, K., RASTI, R., RATNASAMY, S., SHENKER, S., AND CHUN, B.-G. Making sense of performance in data analytics frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (2015), pp. 293–307.

[38] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The pagerank citation ranking: Bringing order to the web.

[39] PETRONI, F., QUERZONI, L., DAUDJEE, K., KAMALI, S., AND IACOBONI, G. Hdrf: Stream-based partitioning for power-law graphs.

[40] QUAMAR, A., DESHPANDE, A., AND LIN, J. Nscale: neighborhood-centric analytics on large graphs. *Proceedings of the VLDB Endowment 7*, 13 (2014), 1673–1676.

[41] ROY, A., BINDSCHAEDLER, L., MALICEVIC, J., AND ZWAENEPOEL, W. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM, pp. 410–424.

[42] ROY, A., MIHAILOVIC, I., AND ZWAENEPOEL, W. X-stream: edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 472–488.

[43] SALIHOGLU, S., AND WIDOM, J. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management* (2013), ACM, p. 22.

[44] SEO, S., YOON, E. J., KIM, J., JIN, S., KIM, J.-S., AND MAENG, S. Hama: An efficient matrix computation with the mapreduce framework. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on* (2010), IEEE, pp. 721–726.

[45] SHAO, B., WANG, H., AND LI, Y. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), ACM, pp. 505–516.

[46] SHI, J., YAO, Y., CHEN, R., CHEN, H., AND LI, F. Fast and concurrent rdf queries with rdma-based distributed graph exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, Nov. 2016), USENIX Association.

[47] SHUN, J., AND BLELLOCH, G. E. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 135–146.

[48] STANTON, I., AND KLIOT, G. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining* (2012), ACM, pp. 1222–1230.

[49] SUNDARAM, N., SATISH, N., PATWARY, M. M. A., DULLOOR, S. R., ANDERSON, M. J., VADLAMUDI, S. G., DAS, D., AND DUBEY, P. Graphmat: High performance graph analytics made productive. *Proc. VLDB Endow. 8*, 11 (July 2015), 1214–1225.

[50] TEIXEIRA, C. H., FONSECA, A. J., SERAFINI, M., SIGANOS, G., ZAKI, M. J., AND ABOULNAGA, A. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM, pp. 425–440.

[51] THAKUR, R., RABENSEIFNER, R., AND GROPP, W. Optimization of collective communication operations in mpich. *International Journal of High Performance Computing Applications 19*, 1 (2005), 49–66.

[52] UGANDER, J., KARRER, B., BACKSTROM, L., AND MARLOW, C. The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503* (2011).

[53] VALIANT, L. G. A bridging model for parallel computation. *Communications of the ACM 33*, 8 (1990), 103–111.

[54] WANG, K., XU, G., SU, Z., AND LIU, Y. D. Graphq: Graph query processing with abstraction refinementscalable and programmable analytics over very large graphs on a single pc. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (2015), pp. 387–401.

[55] WU, M., YANG, F., XUE, J., XIAO, W., MIAO, Y., WEI, L., LIN, H., DAI, Y., AND ZHOU, L. G ra m: scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (2015), ACM, pp. 408–421.

[56] YUAN, P., ZHANG, W., XIE, C., JIN, H., LIU, L., AND LEE, K. Fast iterative graph computation: a path centric approach. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for* (2014), IEEE, pp. 401–412.

[57] ZHANG, K., CHEN, R., AND CHEN, H. Numa-aware graph-structured analytics. In *Proc. PPoPP* (2015).

[58] ZHANG, M., WU, Y., CHEN, K., QIAN, X., LI, X., AND ZHENG, W. Exploring the hidden dimension in graph processing. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, Nov. 2016), USENIX Association.

[59] ZHENG, D., MHEMBERE, D., BURNS, R., VOGELSTEIN, J., PRIEBE, C. E., AND SZALAY, A. S. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (Santa Clara, CA, Feb. 2015), pp. 45–58.

[60] ZHU, X., HAN, W., AND CHEN, W. Gridgraph: Large scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX ATC* (2015).