Electronic Preprint for Journal of Information Processing Vol.29

Regular Paper

An Efficient and Scalable Distributed Hypergraph Processing System

Shugo Fujimura[1,a)] Shigeyuki Sato[1,b)] Kenjiro Taura[1]

Received: April 8, 2021, Accepted: July 27, 2021

Abstract: The recent increase in the amount of graph data has drawn our attention to distributed graph processing systems scalable to large-scale inputs. Although distributed-memory processing is generally less efficient than shared-memory processing because of the communication costs and program complexity, state-of-the-art distributed graph processing systems, such as Gemini, have achieved a comparable efficiency by using lightweight graph partitioning and load balancing. However, the achievement of both scalability and efficiency in hypergraph processing remains an open issue because distributed hypergraph processing systems have not been extensively studied. In this paper, we propose a distributed hypergraph processing system based on Gemini that achieves both scalability and efficiency. Our system outperformed the state-of-the-art shared-memory hypergraph processing system Hygra from several folds to tens of folds on a single-node computer. In addition, it showed speedup in processing a large-scale dataset on a multi-node computer.

Keywords: hypergraph, bipartite graph, distributed graph processing

1. Introduction

With the rapid growth of real-world graph data, particularly on the Web, the demand for analyzing large-scale graph data has been increasing in various fields. In response to this demand, graph processing systems, which are easily capable of graph processing, such as web graph analytics, have been actively studied for more than a decade [8], [18], [19] since the emergence of Google's Pregel [17].

Although many graph processing systems are well-equipped to deal with typical graph analytics, such systems have yet to be developed for advanced graph analytics. A typical instance of advanced analytics is community-level analyses of web graphs. The relationships among communities (e.g., people of the same affiliation and consumers purchasing the same products) in real-world graph data can be naturally modeled as hyperedges, and thus, hypergraph algorithms can solve community-level analyses efficiently. However, for such hypergraph processing, only a few hypergraph processing systems [9], [10], [11], [20] have been developed.

A state-of-the-art hypergraph processing system is Hygra [20], which is an extension of Ligra [22] with a well-designed API for hypergraphs. It has been designed to benefit from Ligra's optimization technique based on the vertex activity and is thus simple yet so efficient to be able to achieve hypergraph processing orders of magnitudes faster than the prior hypergraph processing systems [9], [10], [11]. However, Ligra, the underlying system of Hygra, is a shared-memory (or centralized) graph processing system, and therefore, the input data size is bounded by the memory of a single computer. To handle large-scale input data, we need distributed-memory (or distributed) graph processing systems, which distribute the input data among computing nodes and process the distributed data in parallel.

Distributed graph processing systems are generally less efficient than their centralized counterparts because of the communication costs. However, Gemini [25], a state-of-the-art distributed graph processing system, offers lightweight graph partitioning and communication optimization based on the vertex activity, which brought performance gain sufficient to go beyond the communication overhead and outperformed Ligra even on single computers. We therefore consider the design and implementation of Gemini to be extremely promising for hypergraph processing dealt with by Hygra.

In this paper, we develop a distributed hypergraph processing system by extending Gemini and adding a hypergraph API based on Hygra. The proposed system is implemented by following the design of Gemini to benefit directly from its implementation techniques. The system thus inherits from Gemini the high efficiency and the scalability in terms of the input size. Our experimental evaluation shows that the system significantly outperformed Hygra for all the seven applications used in Ref. [20] on a single-node computer and achieved speedup for a large dataset on a multi-node computer. The proposed system is simple yet capable of efficient distributed hypergraph processing, and therefore, it deserves a new baseline for distributed hypergraph processing systems.

Our main contributions are summarized as follows.

- We have developed a highly efficient distributed hypergraph processing system by extending the state-of-the-art

[1] Graduate School of Information Science and Technology, The University of Tokyo, Bunkyo, Tokyo 113–8656, Japan
[a)] shugo256@eidos.ic.i.u-tokyo.ac.jp
[b)] sato.shigeyuki@mi.u-tokyo.ac.jp

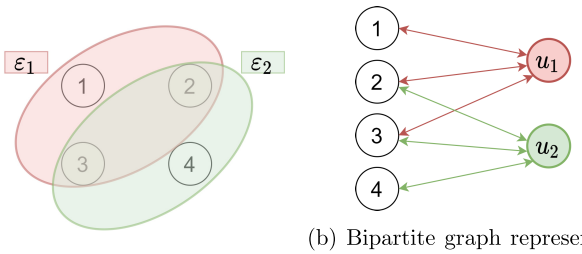
distributed graph processing system Gemini [25] with a hypergraph API based on Hygra [20] (Section 4). The proposed system is designed to benefit from the implementation techniques of Gemini.

- We have experimentally evaluated the proposed system on both single- and multi-node computers with all the seven applications used in Ref. [20] (Section 5). The proposed system achieved several to tens of times higher performance than that of the state-of-the-art centralized hypergraph processing system Hygra for all the applications in a single-node setting and achieved speedup for a large dataset in a multi-node setting.

2. Graphs and Hypergraphs

A graph $G = (V, E)$ consists of a vertex set V and an edge set $E \subseteq V \times V$. In this paper, we assume that $V \subset \mathbb{N}$ and identify vertices with vertex numbers. Unless otherwise noted, graphs refer to directed graphs. For a vertex $v \in V$, $deg^+(v)$ denotes the outdegree of v.

Hypergraphs are a generalization of graphs from one-to-one edges to many-to-many ones. Formally, a hypergraph $H = (V, \mathcal{E})$ consists of a vertex set V and a hyperedge set $\mathcal{E} \subseteq \mathcal{P}(V) \times \mathcal{P}(V)$, where $\mathcal{P}(V)$ denotes the power set of V.



(a) Hypergraph (b) Bipartite graph representation

Fig. 1 Example of bipartite graph representation of a hypergraph. Hyperedge ϵ_i corresponds to vertex u_i, i.e., $\eta(\epsilon_i) = u_i$.

Although hypergraphs are a generalization of graphs, we can represent them with bipartite graphs by interpreting each hyperedge as a vertex. Formally, a bipartite graph $G_b = (V, U, E_f, E_b)$ is a graph consisting of vertex sets U and V, and edge sets $E_f \subseteq V \times U$ and $E_b \subseteq U \times V$ such that $U \cap V = \emptyset$. Now, for a given hypergraph $H = (V, \mathcal{E})$, assuming a one-to-one mapping $\eta : \mathcal{E} \to U$, letting $E_f = \{(v, \eta(\epsilon)) \mid (S, T) = \epsilon \in \mathcal{E} \wedge v \in S\}$ and $E_b = \{(\eta(\epsilon), v) \mid (S, T) = \epsilon \in \mathcal{E} \wedge v \in T\}$, the hypergraph $H = (V, \mathcal{E})$ is isomorphic to the bipartite graph $G_b = (V, U, E_f, E_b)$, i.e., $H \equiv G_b$. This G_b is called the bipartite graph representation of H. **Figure 1** illustrates a concrete example.

In this paper, following the previous study [20], we always treat hypergraphs in the bipartite graph representation: For a hypergraph $H = (V, U, E_f, E_b)$, we simply call $v \in V$ vertices, $u \in U$ hyperedges, and $e \in E_f \cup E_b$ edges.

3. Distributed Graph Processing System Gemini

Figure 2 illustrates an overview of the distributed graph processing system Gemini [25]. Gemini partitions a given graph $G = (V, E)$ by applying chunk-based partitioning (Section 3.2) to the vertex set V. When the system accesses vertices not in local partitions, communication among partitions occurs (Section 3.3). Within a partition, a portion, i.e., a sub-partition, is assigned to each NUMA node by recursively applying chunk-based partitioning. Within each NUMA node, the system divides a sub-partition in terms of tasks rather than data, assigns resultant mini-chunks to each thread, and employs fine-grained work stealing among the threads (Section 3.4) for load balancing.

The following subsections describe the elements of the system.

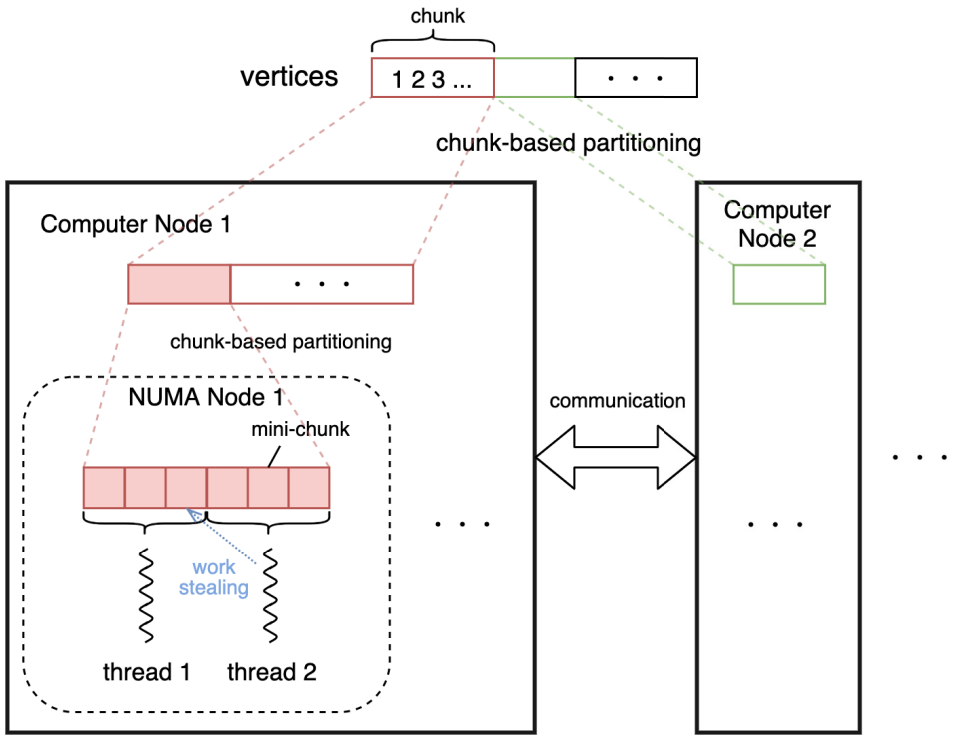
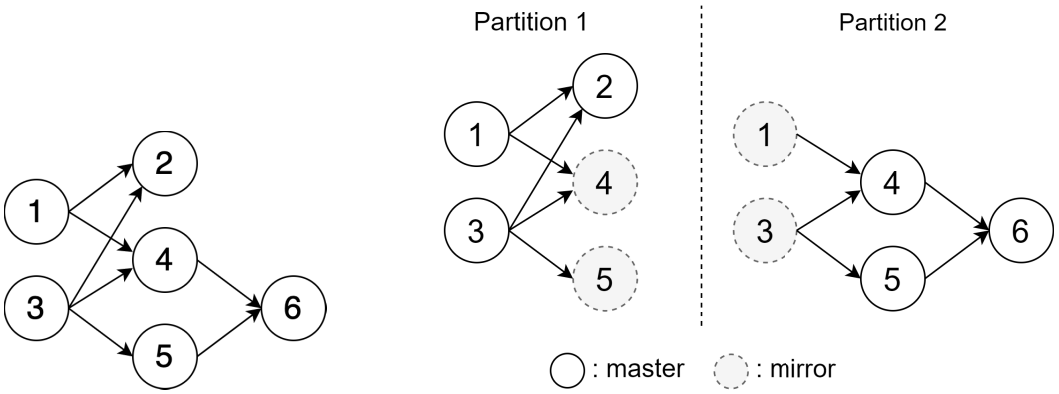


Fig. 2 Overview of Gemini.

Table 1 The primary graph API of Gemini. Type A is overloaded for the operator + and its identity element 0.

Method	Type
`Graph::process_vertices`	$(VertexId \to A) \times VertexBitmap \to A$
`Graph::process_edges`	$(VertexId \to Void) \times (VertexId \times Msg \times OutEdges \to A)$ $\times (VertexId \times InEdges \to Void) \times (VertexId \times Msg \to A) \times VertexBitmap \to A$
`Graph::emit`	$(VertexId \times Msg) \to Void$



(a) Input graph (b) Partitioning based on the master-mirror model

Fig. 3 Illustration of the master-mirror model.

3.1 Gemini API

Gemini provides the graph API through the `Graph` class, which collects the graph structure and graph operations. **Table 1** summarizes the main graph operations provided through the `Graph` class.

First, it is notable that the `Graph` class holds vertex numbers *VertexId*, but not vertex data, i.e., the data associated with vertices. Whereas graph processing systems generally provide a graph data type to hold vertex data, Gemini is designed to manage vertex data outside of the `Graph` class. Instead, it provides an API to allocate arrays of any type indexed by *VertexId*. By contrast, edge data, i.e., the data associated with edges, are managed by the `Graph` class and are immutable (possibly void) in graph operations.

There are two graph operations provided by the `Graph` class: `process_vertices` and `process_edges`. Both are higher-order functions that take functions expected to have side effects not expressed in types.

`process_vertices` takes as the first parameter a function (called a vertex function) that defines the computation on a vertex, and as the second parameter a bitset *VertexBitmap* that selects a subset of vertices; then it applies the vertex function to each of the selected vertices. *VertexBitmap* is designed to be destructively updated by side effects. `process_vertices` in itself does not update the `Graph` object but simply returns the aggregation with + of the return values of a given vertex. Side effects based on *VertexId* to update the vertex data in the vertex function are assumed.

`process_edges` has five parameters. Functions of the first and third parameters, called signal functions, are responsible for sending messages. These messages have to be sent by using the `emit` method of the `Graph` class. Functions of the second and fourth parameters, called slot functions, define the computation for a received message *Msg*. As in vertex functions, slot functions are supposed to have side effects based on *VertexId*, and the aggregation with + of their return values is returned by `process_edges`. *VertexBitmap* of the fifth parameter is a bitset used to select vertices to be the targets of the signal functions. The pair of he first and second parameters are used in the sparse mode and that of the third and fourth parameters are used in the dense mode. The details of the sparse and dense modes will be described in Section 3.3.

The `Graph` class does not provide operations for vertex or edge deletion. Instead, it is designed to use logical deletion with a bitset used to select vertices. `Graph` objects are logically immutable because the vertex data are managed outside the objects and the edge data are immutable.

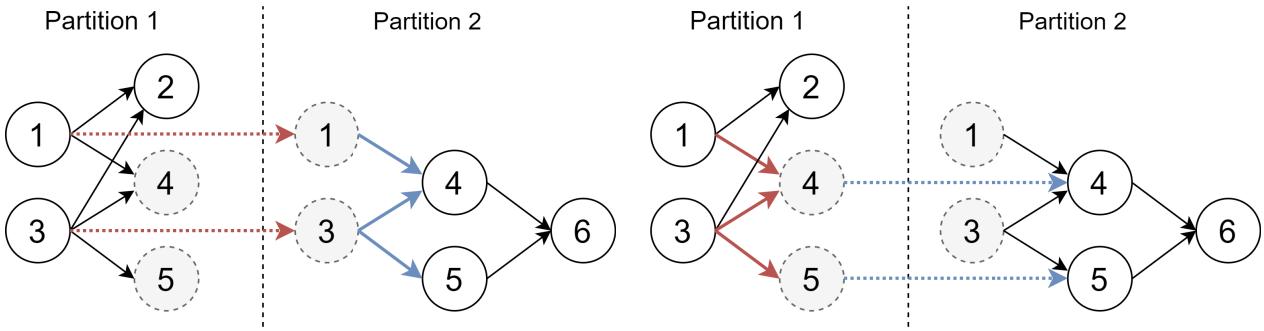
3.2 Graph Partitioning

Gemini divides a given graph into partitions and process them in a distributed manner. It utilizes chunk-based partitioning in the master-mirror model. In this model, the replicas (called mirrors) of vertices outside a partition are logically placed in the partition such that all the outgoing edges of the vertices (called masters) actually inside the partition exist therein. We thus become free from concern about partition boundaries in a vertex-centric view.

Figure 3 illustrates an example of partitioning based on the master-mirror model. When an input graph, as shown in Fig. 3 (a), is partitioned into two sets of master vertices, $\{1, 2, 3\}$ and $\{4, 5, 6\}$, it results in the two partitions shown in Fig. 3 (b).

Vertices 4 and 5 have masters in partition 2 and mirrors in partition 1 because they are adjacent to vertices 1 and 3, which have masters in partition 1. Conversely, vertices 1 and 3 have a mirror in partition 2. The mirror is a dummy vertex in the model, not the actual data stored in memory. Mirrors play the role of endpoints in communication with their masters.

Chunk-based partitioning refers to the construction of the master set of each partition by using a contiguous interval (called a chunk) of vertex numbers. In Fig. 3 (b), each master set is a chunk consisting of sequentially numbered vertices, and thus also an ex-



(a) Sparse mode (b) Dense mode

Fig. 4 Communication method of `process_edges`. The red arrows show the data flow of the signal function, and the blue arrows show the data flow of the slot function.

ample of chunk-based partitioning.

Although various criteria for chunking vertex sets are possible, the balance between load balancing and locality is particularly important. For example, if we divide an input into partitions that have almost the same number of edges, the workloads of the partitions would be nearly the same. However, if the number of mirror nodes increases as a result, the locality would deteriorate, causing more communication.

The chunking strategy of Gemini is to equalize $\alpha|V_i| + |E_i|$, taking account of both the number of vertices $|V_i|$ and that of edges $|E_i|$ in the partition. Here, the coefficient α is a tuning parameter in essence but in Gemini, is empirically determined to be $\alpha = 8(p-1)$, where p denotes the number of partitions.

This chunk-based partitioning can be recursively applied; Gemini also performs NUMA-aware data partitioning on a single computer. Specifically, a partition assigned to a single NUMA computer is divided into sub-partitions and assigned to each of its NUMA nodes. It improves the locality of the computation within the partition.

Chunk-based partitioning was reported by the authors of Gemini to be simple yet surprisingly effective [25] and is a key design choice of Gemini for efficiency.

3.3 Sparse/Dense Mode

There are roughly two ways to send messages between masters and mirrors: For one, masters send messages to their mirrors in the destination partition, and then the data is aggregated in the destination partition, and for the other, mirrors in the source partition aggregate the data and send messages to their masters in an accumulative manner.

In Gemini, the former is called sparse mode, and the latter is called dense mode. **Figure 4** shows a concrete example of each mode.

It is difficult to state whether the sparse mode or dense mode is generally more efficient. For example, if a mirror in the source partition receives data from many different masters, running in the sparse mode would increase the number of inter-partition messages and results in inefficiency. Meanwhile, when running in the dense mode, each endpoint (of masters) has to be able to receive multiple inter-partition messages. This message handling is less efficient than that of the sparse mode, where each endpoint (of mirrors) receives at most one message. It depends on the data flow at runtime.

Gemini switches the communication method adaptively to/from the dense mode and sparse mode. The dense mode is used when the number of edges to be processed in a partition is large, and the sparse mode is used when it is small. Considering that the number of edges to be processed can approximate the amount of data coming into the mirrors in a partition, it avoids the aforementioned disadvantage of the sparse mode.

The number of edges to be processed in a partition can be determined based on an argument *VertexBitmap* passed to `process_edges` and the graph structure. Specifically, letting $B \subseteq V$ be an argument of *VertexBitmap*, if $|B| + \sum_{v \in B} deg^+(v)$ is large, it will run in the dense mode, and if it is small, it will run in the sparse mode.

The number of edges to be processed expresses the activity of the partition. Therefore, the adaptive switching of the sparse/dense mode is an adaptation of the idea of the optimization technique based on the vertex activity of Ligra [22] to distributed graph processing.

3.4 Fine-grained Work Stealing

As described in Section 3.2, Gemini applies two-level chunk-based partitioning to an input graph to distribute the data to NUMA node units. However, within a NUMA node, for the sake of load-balancing quality, Gemini does not further apply recursive partitioning to distribute the data into cores but instead performs work stealing of fine-grained tasks.

First, the vertex sequence (i.e., sub-partition) assigned to each NUMA node is divided into small chunks of fixed length (64 by default) called mini-chunks. The sequence of mini-chunks is then divided into blocks and assigned to each thread. Each thread processes tasks per mini-chunk. When a thread has finished the assigned mini-chunks, it steals and processes a mini-chunk from another thread. The victim thread of this work stealing is selected in a round-robin fashion.

4. The Proposed System

In this section, we describe the design and implementation of our distributed hypergraph processing system.

The basic design aims to maintain a hypergraph $H = (V, U, E_f, E_b)$ by using Gemini as the underlying implementation and to provide a hypergraph API based on Hygra [20]. We aim to

Table 2 The primary hypergraph API of the proposed system. Let $H = (V, U, E_f, E_b)$ be an input hypergraph.

Method	Description
`Hypergraph::process_vertices`	Applies `Graph::process_vertices` to graph $(V, E_f \cup E_b)$
`Hypergraph::process_hyperedges`	Applies `Graph::process_vertices` to graph $(U, E_f \cup E_b)$
`Hypergraph::prop_from_vertices`	Applies `Graph::process_edges` to graph $(V \cup U, E_f)$
`Hypergraph::prop_from_hyperedges`	Applies `Graph::process_edges` to graph $(V \cup U, E_b)$
`Hypergraph::filter_vertices_from_hyperedges`	Given predicate p and $U' \subseteq U$, returns $\{v \mid (u, v) \in E_b \wedge p(u) \wedge u \in U'\}$.
`Hypergraph::filter_hyperedges_from_vertices`	Given predicate p and $V' \subseteq V$, returns $\{u \mid (v, u) \in E_f \wedge p(v) \wedge v \in V'\}$.

preserve the high efficiency of Gemini by reusing its implementation.

4.1 API

The core of the extension to Gemini is the `Hypergraph` class, which provides the hypergraph API for the bipartite graph representation. The main hypergraph operations are summarized in **Table 2**.

The four hypergraph operations, `process_vertices`, `process_hyperedges`, `prop_from_vertices`, and `prop_from_hyperedges`, are operations that focus on each component of a given hypergraph (V, U, E_f, E_b). As described in Table 2, each operation is implemented by reusing the API implementation of Gemini almost directly.

The filter operation `Hypergraph::filter_vertices_from_hyperedges` returns the vertex set V' adjacent to the hyperedge set selected by a given predicate. It is an operation to logically delete the vertex set $V \setminus V'$. Since `Hypergraph` is an immutable data structure like `Graph`, it returns a bitset representing V' without changing the graph structure. The logical deletion is achieved by feeding this bitset to the subsequent hypergraph operation. This filter operation is a high-level wrapper for `prop_from_hyperedges` that uses the built-in signal/slot functions. Thus, as in the basic operations, it takes a bitset U' of U that selects the processing targets (in this case, the targets of the predicate). In addition, as an optional parameter, it takes the array of logical degrees of U, deg_U, and decreases the logical degree according to the logical deletion. The hyperedge set $\{u \in U \mid deg_U[u] = 0\}$, whose logical degree is zero, can be logically deleted by using `Hypergraph::process_hyperedges`. `Hypergraph::filter_vertices_from_hyperedges` is a symmetric operation to `Hypergraph::filter_vertices_from_hyperedges`.

The `Hypergraph` class provides supplementary data on the hypergraph (V, U, E_f, E_b) as follows:

- The number of vertices $|V|$, that of hyperedges $|U|$, and that of edges $|E_f| + |E_b|$;
- The indegrees and outdegrees for every $v \in V$ and $u \in U$.

As in `Graph` of Gemini, the data corresponding to vertices $v \in V$ and hyperedges $u \in U$ are designed to be managed outside of the `Hypergraph` class. `Hypergraph` provides methods `alloc_vertex_array` and `alloc_hyperedge_array` for allocating arrays of any element type, respectively. The memory allocation is designed to be NUMA-aware, following the array allocation of Gemini. `Hypergraph` can also hold immutable data corresponding to the edges $e \in E_f \cup E_b$.

Bitsets for selecting the target of the signal/slot functions are of the same data type as *VertexBitmap* of Gemini. Bitsets for V and U can be constructed with the methods `alloc_vertex_subset` and `alloc_hyperedge_subset`, respectively.

Listing 1 Implementation of SSSP by the proposed system

```
void sssp(Hypergraph<Weight>& h,
          VertexId src) {
  // Initialize activity bitmaps
  auto active_V = h.alloc_vertex_subset();
  active.set_bit(src);
  auto active_U =
    h.alloc_hyperedge_subset();

  // Initialize vertex/hyperedge distance
  auto d_V = h.alloc_vertex_array();
  d_V.fill(INFTY);
  d_V[src] = 0;
  auto d_U = h.alloc_hyperedge_array();
  d_U.fill(INFTY);

  // Frontier-based iteration
  while (active_V.size > 0) {
    active_U.clear();
    h.prop_from_vertices(
      sparse_signal_gen<VertexId>(h, d_V),
      sparse_slot_gen<VertexId>(h, d_U,
                                active_U),
      dense_signal_gen<HyperedgeId>(h, d_V),
      dense_slot_gen<HyperedgeId>(h, d_U,
                                  active_U),
      active_V);

    active_V.clear();
    h.prop_from_hyperedges(
      sparse_signal_gen<HyperedgeId>(h, d_U),
      sparse_slot_gen<HyperedgeId>(h, d_V,
                                   active_V),
      dense_signal_gen<VertexId>(h, d_U),
      dense_slot_gen<VertexId>(h, d_V,
                               active_V),
      active_U);
  }
}
```

4.2 Example Use: Single Source Shortest Path Problem

Listings 1 and 2 show an example implementation of the single source shortest path problem (SSSP) by using our system. The input is a hypergraph in the bipartite graph representation, and distances are assigned to edges.

The function `sssp` implements an algorithm equivalent to the one used in the previous study [20]. It updates the shortest distance of each vertex in an iterative manner while managing a set of vertices to be processed, called a frontier.

Initially, put only the source vertex, `src`, into the frontier (i.e., set the corresponding bit in `active_V`) and initialize the distance to 0. Initialize the other distances $v \in V$ and $u \in U$ to ∞ (i.e., `INFTY`). The distances of each vertex and hyperedge are kept in the arrays `d_V` and `d_U`, respectively. Then, `prop_from_vertices` and `prop_from_hyperedge` are alternately invoked to propagate the shortest distance over the hyperedges until the frontier becomes empty. The slot functions update the frontier by setting bits of `active_V` and `active_U` for $v \in V$ and $u \in U$ of which distances have been updated.

Listing 2 Signal/slot functions for SSSP

```
template <typename SrcId>
auto sparse_signal_gen(Hypergraph<Weight>& h,
                       Array<Weight>& d) {
  return [&](SrcId v) { h.emit(v, d[v]); };
}

template <typename SrcId>
auto sparse_slot_gen(Hypergraph<Weight>& h,
                     Array<Weight>& d,
                     Bitmap& active) {
  return [&](SrcId v,
             Weight msg,
             OutEdges<Weight> edges) {
    for (auto e : edges) {
      auto relax_dist = msg + e.data;
      if (relax_dist < d[e.dst]) {
        atomic_update(&d[e.dst], relax_dist);
        active.set_bit(e.dst);
      }
    }
  };
}

template <typename DstId>
auto dense_signal_gen(Hypergraph<Weight>& h,
                      Array<Weight>& d) {
  return [&](DstId u,
             InEdges<Weight> edges) {
    auto msg = INFTY;
    for (auto e in edges) {
      msg = std::min(msg, d_V[e.src]+e.data);
    }
    if (msg < INFTY) h.emit(u, msg);
  };
}

template <typename DstId>
auto dense_slot_gen(Hypergraph<Weight>& h,
                    Array<Weight>& d,
                    Bitmap& active) {
  return [&](DstId dst, Weight msg) {
    if (msg < d[dst]) {
      atomic_update(&d[dst], msg);
      active.set_bit(dst);
    }
  };
}
```

`prop_from_vertices` and `prop_from_hyperedge`, which propagate the shortest distance, use signal/slot functions symmetric with respect to vertex U and hyperedge V. To take advantage of this symmetry, the functions in Listing 2, are defined such that they parameterize a target type (`VertexId` or `HyperedgeId`), take a bitset and distance array for V or U, and yield lambda expressions of the signal/slot functions.

4.3 Advantages of the Hypergraph API

Since the bipartite graph representation of a hypergraph is merely a graph, it is possible to obtain the same results only by using the Gemini API (Table 1), without using the API of the proposed system (Table 2). In fact, for the SSSP example above, the results would be the same even if you interpret the bipartite graph representation as a regular graph and use the `Graph` class to perform the SSSP. However, there are two advantages to providing the hypergraph API.

One advantage is the clarity of the program description. For example, to scan V and U alternately supposing the bipartite graph representation, it is necessary to prepare bitsets representing V and U and give them appropriately. Furthermore, when keeping and updating different data for vertex V and hyperedge U, the Gemini API necessitates manual management whether $x \in V$ or $x \in U$ for the target vertex x of the vertex function or slot/signal function. For example, in the case of SSSP, the API can be used by being aware of the even/odd number of invocations of `process_edges`, although doing so is clearly error-prone. Above all, the hypergraph algorithm is clearer if it is written with an API that explicitly, rather than implicitly, handles hypergraphs.

The other advantage is to provide room for optimization specific to hypergraph processing. In real-world data, vertices and hyperedges model different types of data. For example, in e-commerce data, we consider modeling the buyers as vertices V and the purchased products as hyperedges U. The number of buyers $|V|$ and that of products $|U|$ are naturally different. The degree distributions of the relationships of purchasing and being purchased, that is, $(V, E_f \cup E_b)$ and $(U, E_f \cup E_b)$, are also naturally different. Consequently, there would be a natural difference in locality and activity at runtime. If V and U are not distinguished at the system level, it is hard for the system to provide data partitioning, communication, and scheduling that are aware of the differences of the properties of V and U. Therefore, telling the system that the data is a hypergraph through the API brings a better synergy between the programmer and system.

4.4 Adaptation to Gemini

The proposed processing system is basically implemented by adding the `Hypergraph` class to Gemini. The internal implementation of the `Hypergraph` class substantially manages (V, E_f) and (U, E_b) in `Graph`. In other words, the Gemini implementation, including chunk-based partitioning, communication mode switching, and fine-grained work stealing, is reused as-is to construct (V, U, E_f, E_b).

This approach is simple yet reasonable in terms of locality. In the proposed system, the data for V and U are never accessed together because of the API design. Even in `prop_from_vertices`, which can access both of them, the signal and slot functions are called in different stages, and thus, the data access of V in the signal function and the data access of U in the slot function are separated in time. By symmetry, the same is true for `prop_from_hyperedges`. Therefore, managing (V, E_f) and (U, E_b) independently in a single partition increases locality.

However, the partition obtained by pairing independent chunk-based partitions of (V, E_f) and (U, E_b) may not be advantageous in terms of the locality of inter-partition access (i.e., communication). In chunk-based partitioning, a contiguous interval of vertices (i.e., a chunk) is assigned to a partition, assuming that the natural order of vertices represents the natural locality well. Even if the hypergraph $(V, \mathcal{E})$ has locality with respect to the natural order of V, the relation between the i-th chunk V_i of V and the i-th chunk U_i of U does not necessarily have locality. In other words, it is not necessarily the case that their pair (V_i, U_i) constitutes a better partition than the other combinations.

In this study, we assume that the locality of natural orderings assumed in chunk-based partitioning is also valid for the bipartite graph representations of hypergraphs, and adopt the policy of constructing partitions with (V_i, U_i). For both V and U partitions, we adopted the same α criterion as in Gemini. We leave the investigation of the validity of this approach for future work.

Finally, we note some implementation details. The `Hypergraph` class is implemented by reusing most of the internal implementation of the `Graph` class, but does not actually use an instance of `Graph`; instead, it is implemented by retaining (V, U, E_f, E_b) as data and using the code of the methods of `Graph` according to the interpretation of the hypergraph API based on `Graph`.

4.5 API Comparison to Hygra

The proposed system extends Gemini on the basis of the hypergraph API of Hygra [20]. The API correspondence is summarized in **Table 3**. Because the API of the proposed system is designed as an extension of Gemini, it is not fully compatible at the interface level, but cover the primary hypergraph API of Hygra at the functionality level. Here, we describe the differences between them.

As for VertexMap and HyperedgeMap, the proposed API corresponds directly to the Hygra API. VertexProp and HyperedgeProp are operations for selecting a portion of the hyperedge set U and vertex set V along E_f and E_b, respectively. The corresponding `prop_from_vertices` and `prop_from_hyperedges` follow `process_edges` of Gemini and are designed to take signal/slot functions and update data related to the selected object. This allows `prop_from_hyperedges` to include the functionality of HyperedgePropCount, which processes V along with the degrees on (V, E_b). For HyperedgeFilterNgh, it supports the filter function but differs in that it destructively updates the hypergraph. This is due to the difference in the way the graph data are represented between Ligra, upon which Hygra is based, and Gemini, upon which the proposed system is based.

Although Hygra also provides a bucketing API based on Julienne [5], the proposed processing system does not. It is a design choice based on the technical scope of this study, and more concretely, is due to the following three points.

- Bucketing is not a hypergraph-specific operation.
- Bucketing was only used to improve the efficiency of k-core decomposition in the seven applications covered in Hygra [20].
- All the existing systems that provide bucketing APIs [5], [20], [24] are centralized, not distributed, graph processors.

In other words, bucketing is an auxiliary operation in hypergraph processing, and whether it is useful to implement in distributed graph processing remains unclear. Because our main focus is to construct an efficient distributed hypergraph processing system by extending Gemini for hypergraphs, we have judged that implementing bucketing in Gemini is beyond the scope of the present study.

Table 3 API correspondence between Hygra [20] and the proposed system.

Hygra	Proposed System (`Hypergraph::*`)
VertexMap	`process_vertices`
HyperedgeMap	`process_hyperedges`
VertexProp	`prop_from_vertices`
HyperedgeProp	`prop_from_hyperedges`
HyperedgeFilterNgh	`filter_vertices_from_hyperedges`
HyperedgePropCount	`prop_from_hyperedges`

5. Evaluation

We experimentally evaluate the proposed system for the following three points:

- Whether it achieves efficient hypergraph processing on the basis of the implementation techniques of Gemini (Section 5.2.1);
- Whether it achieves distributed hypergraph processing scalable for large-scale input (Section 5.2.2);
- Whether its specialized implementation to hypergraph processing works effectively on top of Gemini (Section 5.2.3).

5.1 Experimental Settings

By using the proposed system, we implemented the following seven applications that had been used as benchmarks in Hygra [20].

- Betweenness Centrality (BC)
- HyperTree (BFS)
- Connected Components (CC)
- K-core Decomposition (KC)
- Maximum Independent Set (MIS)
- Page Rank (PR)
- Single Source Shortest Path (SSSP)

When comparing ours with Gemini, for BC, BFS, CC, and SSSP, the benchmark applications bundled with Gemini were used as-is; for KC, MIS, and PR, we implemented the counterparts that performed equivalent computations on bipartite graphs only with the Gemini API.

The proposed system and the benchmark applications are available online: https://github.com/shugo256/GeminiGraph/tree/hypergraph.

For the input datasets, we selected four hypergraphs from the benchmarks [21] provided with Hygra (**Table 4**).

Com-orkut and friendster are graphs with community data, which were originally provided in the Stanford Large Network Dataset Collection [15] and were converted into hypergraphs by replacing communities with hyperedges. Web is a hypergraph that was made by reinterpreting a bipartite graph in the Koblenz Network Collection [13]. These three are real-world hypergraphs. Note that web is a real-world hypergraph of the largest numbers of vertices and edges in the bipartite graph representation among the ones used in the previous studies [9], [20]. By contrast, rand1 was generated such that each hyperedge contained 10 randomly chosen vertices [20], and is an artificial large-scale hypergraph for demonstrating scalability.

Table 4 Datasets used in the experiments.

Dataset	$\lvert V\rvert$	$\lvert U\rvert$	$\lvert E_f\rvert + \lvert E_b\rvert$
com-orkut [15]	2.32×10^6	1.53×10^7	1.07×10^8
friendster [15]	7.94×10^6	1.62×10^6	2.35×10^7
web [13]	2.77×10^7	1.28×10^7	1.41×10^8
rand1 [20]	1.00×10^8	1.00×10^8	1.00×10^9

Table 5 Specifications of computing nodes.

CPU	Intel® Xeon® Platinum 8280 (2.7 GHz)
#CPUs (#cores)	2 (28 + 28)
Memory	192 GiB
Interconnect	Intel® Omni-Path (100 Gbps)

When comparing ours with Hygra and Gemini, all the datasets were used. By contrast, when evaluating the scalability of distributed processing, only rand1 was used. This is because we have judged that datasets other than rand1 are too small for distributed processing on multi-nodes and are unsuitable for the scalability evaluation.

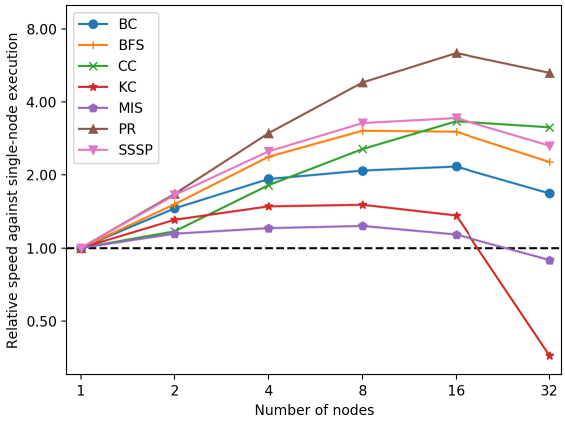


Fig. 5 Scalability in multi-node execution for rand1.

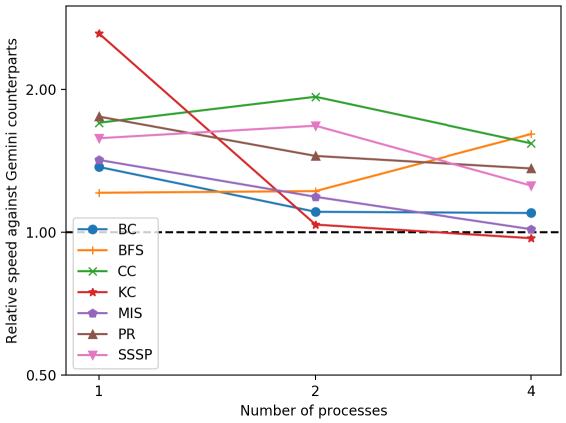
We used Oakbridge-CX, a supercomputing system operated by the Information Technology Center of the University of Tokyo, for the experimental environment. The specifications of its computing nodes are summarized in **Table 5**.

For a fair comparison with Hygra, the number of threads was set to 56, which is equal to the number of cores, with a single process. For comparison with Gemini, to examine the performance of distributed processing on small inputs other than rand1, we set one process per computing node and one thread per process. To evaluate the scalability of distributed processing, we allocated one process per CPU (i.e., NUMA node) running with a single thread.

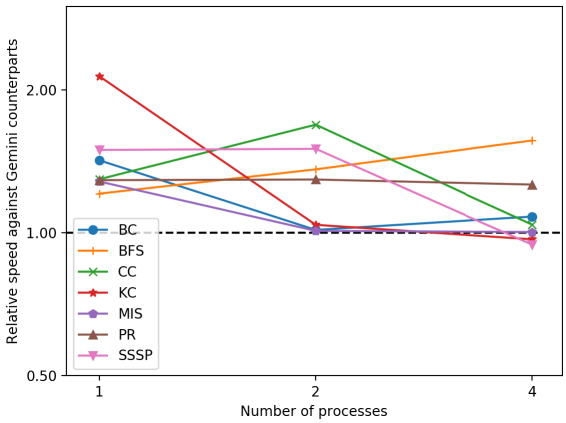
The applications of the proposed system were compiled by using g++ 4.8.5 with the -O3 option. For the MPI communication of Gemini, which is also the base of the proposed system, Intel® MPI Library 2019 Update 9 was used. Hygra applications were compiled by using Intel® C++ Compiler 19.1.3.304 with the -O3

Table 6 Execution time [s] of each application on a single node. Speed ratio to Hygra in parentheses.

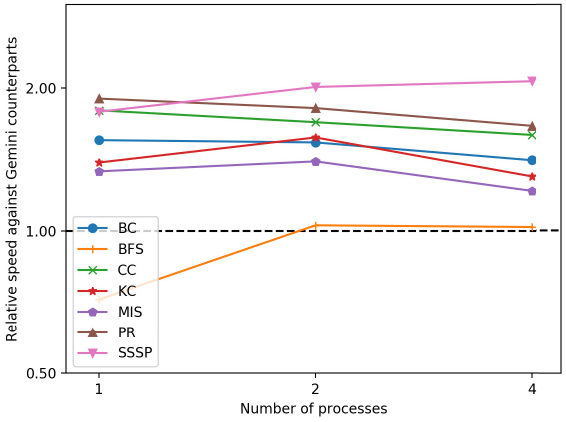
	BC	BFS	CC	KC	MIS	PR	SSSP
com-orkut	0.184 (27.4)	8.96×10^{-2} (9.20)	0.259 (27.8)	1.30 (11.1)	0.853 (44.0)	1.51 (42.7)	0.169 (40.0)
friendster	0.126 (16.3)	5.97×10^{-2} (9.02)	0.179 (18.0)	0.135 (42.2)	0.547 (8.99)	0.703 (25.6)	0.128 (30.2)
web	0.352 (21.4)	0.139 (9.52)	0.326 (25.6)	38.3 (371)	3.63 (7.93)	2.68 (30.3)	0.178 (32.7)
rand1	2.17 (49.2)	1.13 (19.5)	5.17 (56.1)	20.4 (40.2)	9.76 (40.1)	28.6 (46.9)	5.23 (22.0)



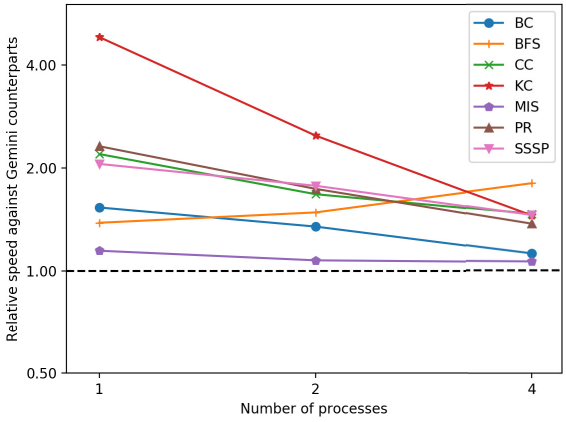
(a) com-orkut



(b) friendster



(c) rand1



(d) web

Fig. 6 Relative speed to Gemini.

option.

For the measurements of each application, we used the median of the execution times of five runs after one warm-up run.

5.2 Experimental Results

5.2.1 Comparison with Hygra

Table 6 summarizes the execution time of each application of the proposed system on a single node. For all the applications used in the experiments, the proposed system outperformed Hygra by several to several dozen times. Gemini, the base of the proposed system, was comparable to Ligra, the base of Hygra, in terms of single-node performance and had been reported to be at most twice as fast as Ligra [25]. Although the proposed system is a simple extension of Gemini, the speed ratio of the proposed system to Hygra was much greater than that of Gemini to Ligra.

On Hygra, a more efficient algorithm for KC (KC-E) was implemented with the bucketing API mentioned in Section 4.5. We also measured the speed ratio of KC of the proposed system to KC-E of Hygra, and the results were 4.80 for com-orkut, 13.6 for friendster, 0.272 for web, and 4.95 for rand1; the proposed system outperformed Hygra except for the web case even without a bucketing API.

These results demonstrate that the proposed system achieves highly efficient hypergraph processing that significantly outperforms Hygra by utilizing the implementation techniques of Gemini.

5.2.2 Scalability

Figure 5 shows the effect of the proposed system on the number of computing nodes for each application. The proposed system achieved multi-node executions by distributing the input data. We observed the speedup up to 8 nodes for BFS, KC, and MIS, and up to 16 nodes for the other four applications.

The results show that the proposed system can achieve distributed processing for large hypergraphs that cannot fit in single-node memory without sacrificing the execution speed through a multi-node execution.

5.2.3 Comparison with Gemini

Figure 6 shows the relative speed of the proposed system with respect to Gemini. The number of processes (or computing nodes) is limited to the range of 1 to 4 because of the small input size.

As seen from Fig. 6, the proposed system generally outperformed Gemini. This indicates that the implementation of the proposed system, which manages and processes V and U separately, is an appropriate specialization for hypergraph processing. The performance differences tended to decrease a little when the number of processes increased. We attribute it to that the performance gain from the specialization became relatively small because of the communication-derived performance differences. We therefore consider that the performance differences in the single-process case are the most accurate explanation of the performance gain of the specialization.

6. Related Work and Discussions

Only a few hypergraph processing systems have been studied. To the best of our knowledge, except for Hygra [20], which were compared in Section 5, only HyperX [10], [11] and MESH [9] were developed. Both of these systems were built on top of Apache Spark [23], a data processing framework for distributed memory environments, and the distributed collections (called RDDs) of Spark were used to represent distributed hypergraphs. MESH was built upon the API of GraphX [7] and aimed at an implementation more compact and simpler than HyperX. There was no significant difference in efficiency between them [9].

HyperX and MESH implemented hypergraph partitioning by using RDDs. However, RDDs themselves do not allow for destructive updates, and graph processing that requires iterative graph updates incurs large overhead at the design level. Unlike Ligra [22] and Gemini [25], the execution strategy based on vertex activity is difficult to implement efficiently, and every iteration ends up traversing all vertices and sending messages. As a result, MESH was reported to fall far short of the performance of Hygra [20]. Our system is technically different from those systems in that it builds upon Gemini to take advantage of the efficiency and scalability derived from the implementation techniques of Gemini.

In this study, although we applied the chunk-based partitioning of Gemini to each of the vertex set and hyperedge set, this partitioning is unaware of the structure (particularly, modularity) of hypergraphs. A parallel partitioning algorithm for large-scale hypergraphs has recently been proposed by Maleki et al. [16], which is promising as a preprocessing method for distributed hypergraph processing. However, the design choice is not obvious because the cost of partitioning itself is also important when incorporated into processing systems, as chunk-based partitioning demonstrated. Hypergraph partitioning for distributed hypergraph processing systems is left for future work.

The reason why chunk-based partitioning works in favor of locality is based on the assumption that the natural order of vertices is a good representation of the natural locality of the graph data. In fact, it was shown on a typical dataset that it was more efficient to preserve the natural order under chunk-based partitioning than to destroy it with hash-based partitioning [25]. However, as noted in Section 4.4, it is not clear that this assumption is directly applicable to hypergraphs in bipartite graph representations. In actual use cases, it is natural to assume that the graph data are modeled as a hypergraph and then given a preprocessing step to convert it into a bipartite graph representation. It is not obvious whether this preprocessing can be implemented as efficiently as chunk-based partitioning while preserving the natural locality.

Another possible direction is to improve the quality of the partitioning by reordering the vertices to better represent the natural locality of the data, without regard to the natural order of the input. Although vertex reordering has been well studied in the context of graph compression [1], [3], the cost of reordering for compression is high, and it is questionable whether it is suitable for preprocessing in graph processing systems. We believe that lightweight graph reordering [2], [6] for shared-memory (centralized) graph processing systems, which has been well studied in recent years, is more promising. Moreover, for pattern mining on web graphs, a lightweight method to compress overlapping link structures into a single virtual node was developed [4]. In a simi-

lar way, we believe that combining overlapping hyperedges into a single virtual node is promising for constructing an efficient distributed hypergraph representation.

In addition to the bipartite graph representation, there is also a representation called clique expansion [9], where each hyperedge is transformed into a clique of the vertices that it contains. This clique representation loses the many-to-many relationship of a hypergraph, and models the data as a simple graph. As a result, it is not possible to recover the original hypergraph from the clique representation without information to distinguish the cliques derived from the hyperedges. Moreover, the number of edges required to represent one hyperedge $\epsilon = (S, T)$ is $|S| + |T|$ in the bipartite graph representation, but is $|S| \cdot |T|$ in the clique representation, which consumes more memory. In fact, the investigation on Hygra [20] showed a significant advantage of the bipartite graph representation in terms of runtime performance. However, dense substructures such as cliques bring high locality and have an overall smaller diameter than bipartite graph representations. There is still a possibility that partial clique representations have an advantage.

Finally, in our experiments, we used a randomly generated dataset as the large-scale hypergraph, following the previous study [20]. However, recent studies [12], [14] have revealed that randomly generated hypergraphs do not necessarily represent real-world datasets. Therefore, an experimental investigation with large datasets based on the real-world hypergraph models proposed therein is important for the design and implementation of distributed hypergraph processing systems.

7. Conclusions

In this study, we have developed an efficient distributed hypergraph processing system that inherits the advantages of Gemini for efficient distributed graph processing. The proposed system significantly outperformed Hygra, a state-of-the-art centralized hypergraph processing system, in single-node execution for all seven applications examined, and also achieved speedup for large-scale data in multi-node execution. The efficient distributed processing by the proposed system provides scalability for large-scale hypergraph processing hard to handle on a single node.

As discussed in Section 6, studies on hypergraph processing systems are still underdeveloped and leave much room for further investigation. In this context, this study provides a new baseline for distributed hypergraph processing systems. We expect that studies on distributed hypergraph processing systems will develop further by incorporating new techniques into the proposed system.

References

[1] Apostolico, A. and Drovandi, G.: Graph Compression by BFS, *Algorithms*, Vol.2, No.3, pp.1031–1044 (online), DOI: 10.3390/a2031031 (2009).

[2] Balaji, V. and Lucia, B.: When is Graph Reordering an Optimization? Studying the Effect of Lightweight Graph Reordering Across Applications and Input Graphs, *2018 IEEE International Symposium on Workload Characterization, IISWC '18*, pp.203–214, IEEE (online), DOI: 10.1109/IISWC.2018.8573478 (2018).

[3] Boldi, P., Rosa, M., Santini, M. and Vigna, S.: Layered Label Propagation: A Multiresolution Coordinate-Free Ordering for Compressing Social Networks, *Proc. 20th International Conference on World Wide Web, WWW '11*, pp.587–596, ACM (online), DOI: 10.1145/1963405.1963488 (2011).

[4] Buehrer, G. and Chellapilla, K.: A Scalable Pattern Mining Approach to Web Graph Compression with Communities, *Proc. 2008 International Conference on Web Search and Data Mining, WSDM '08*, pp.95–106, ACM (online), DOI: 10.1145/1341531.1341547 (2008).

[5] Dhulipala, L., Blelloch, G. and Shun, J.: Julienne: A Framework for Parallel Graph Algorithms using Work-efficient Bucketing, *Proc. 29th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '17*, pp.293–304, ACM (online), DOI: 10.1145/3087556.3087580 (2017).

[6] Faldu, P., Diamond, J. and Grot, B.: A Closer Look at Lightweight Graph Reordering, *2019 IEEE International Symposium on Workload Characterization, IISWC '19*, pp.1–13, IEEE (online), DOI: 10.1109/IISWC47752.2019.9041948 (2019).

[7] Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J. and Stoica, I.: GraphX: Graph Processing in a Distributed Dataflow Framework, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14*, pp.599–613, USENIX Association (2014) (online), available from ⟨https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez⟩.

[8] Heidari, S., Simmhan, Y., Calheiros, R.N. and Buyya, R.: Scalable Graph Processing Frameworks: A Taxonomy and Open Challenges, *ACM Comput. Surv.*, Vol.51, No.3, pp.60:1–60:53 (online), DOI: 10.1145/3199523 (2018).

[9] Heintz, B., Hong, R., Singh, S., Khandelwal, G., Tesdahl, C. and Chandra, A.: MESH: A Flexible Distributed Hypergraph Processing System, *2019 IEEE International Conference on Cloud Engineering, IC2E '19*, pp.12–22, IEEE (online), DOI: 10.1109/IC2E.2019.00-11 (2019).

[10] Huang, J., Zhang, R. and Yu, J.X.: Scalable Hypergraph Learning and Processing, *2015 IEEE International Conference on Data Mining, ICDM '15*, pp.775–780, IEEE (online), DOI: 10.1109/ICDM.2015.33 (2015).

[11] Jiang, W., Qi, J., Yu, J.X., Huang, J. and Zhang, R.: HyperX: A Scalable Hypergraph Framework, *IEEE Trans. Knowl. and Data Eng.*, Vol.31, No.5, pp.909–922 (online), DOI: 10.1109/TKDE.2018.2848257 (2019).

[12] Kook, Y., Ko, J. and Shin, K.: Evolution of Real-world Hypergraphs: Patterns and Models without Oracles, *2020 IEEE International Conference on Data Mining, ICDM '20*, pp.272–281, IEEE (online), DOI: 10.1109/ICDM50108.2020.00036 (2020).

[13] Kunegis, J.: Handbook of Network Analysis – The KONECT Project (2019), available from ⟨https://github.com/kunegis/konect-handbook/raw/master/konect-handbook.pdf⟩.

[14] Lee, G., Choe, M. and Shin, K.: How Do Hyperedges Overlap in Real-World Hypergraphs? – Patterns, Measures, and Generators, *Proc. Web Conference 2021, WWW '21*, arXiv:2101.07480 (2021).

[15] Leskovec, J. and Krevl, A.: SNAP Datasets: Stanford Large Network Dataset Collection (2014), available from ⟨http://snap.stanford.edu/data⟩.

[16] Maleki, S., Agarwal, U., Burtscher, M. and Pingali, K.: BiPart: A Parallel and Deterministic Hypergraph Partitioner, *Proc. 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '21*, pp.161–174, ACM (online), DOI: 10.1145/3437801.3441611 (2021).

[17] Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I. and Czajkowski, N.L.G.: Pregel: A System for Large-Scale Graph Processing, *Proc. 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10*, pp.135–146, ACM (online), DOI: 10.1145/1807167.1807184 (2010).

[18] McCune, R.R., Weninger, T. and Madey, G.: Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing, *ACM Comput. Surv.*, Vol.48, No.2, pp.25:1–25:39 (online), DOI: 10.1145/2818185 (2015).

[19] Shi, X., Zheng, Z., Zhou, Y., Jin, H., He, L., Liu, B. and Hua, Q.-S.: Graph Processing on GPUs: A Survey, *ACM Comput. Surv.*, Vol.50, No.6, pp.81:1–81:35 (online), DOI: 10.1145/3128571 (2018).

[20] Shun, J.: Practical Parallel Hypergraph Algorithms, *Proc. 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '20*, pp.232–249, ACM (online), DOI: 10.1145/3332466.3374527 (2020).

[21] Shun, J.: Practical Parallel Hypergraph Algorithms (PPoPP 2020 Artifact Evaluation) (2020), available from ⟨https://github.com/jshun/ppopp20-ae/⟩.

[22] Shun, J. and Blelloch, G.E.: Ligra: A Lightweight Graph Processing Framework for Shared Memory, *Proc. 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13*, pp.135–146, ACM (online), DOI: 10.1145/2442516.2442530 (2013).

[23] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M.J., Shenker, S. and Stoica, I.: Resilient Dis-

tributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, *9th USENIX Symposium on Networked Systems Design and Implementation, NSDI '12*, pp.15–28, USENIX Association (2012) (online), available from ⟨https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia⟩.

[24] Zhang, Y., Brahmakshatriya, A., Chen, X., Dhulipala, L., Kamil, S., Amarasinghe, S. and Shun, J.: Optimizing Ordered Graph Algorithms with GraphIt, *Proc. 18th ACM/IEEE International Symposium on Code Generation and Optimization, CGO '20*, pp.158–170, ACM (online), DOI: 10.1145/3368826.3377909 (2020).

[25] Zhu, X., Chen, W., Zheng, W. and Ma, X.: Gemini: A Computation-Centric Distributed Graph Processing System, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI '16*, pp.301–316, USENIX Association (2016) (online), available from ⟨https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhu⟩.

Shugo Fujimura is a graduate student in the Graduate School of Information Science and Technology at the University of Tokyo. He received his B.E. from the University of Tokyo in 2021. He is currently studying IoT and networks.

Shigeyuki Sato is an Assistant Professor in the Graduate School of Information Science and Technology at the University of Tokyo. He received his Ph.D. from the University of Electro-Communications in 2015. His research interest is in compilers and parallel programming, especially, automatic parallelization, program synthesis, high-level optimizations, domain-specific languages, parallel patterns, and tree/graph processing. He is also a member of ACM and JSSST.

Kenjiro Taura is a Professor in the Department of Information and Communication Engineering at the University of Tokyo. He received his B.S., M.S., and Ph.D. from the University of Tokyo in 1992, 1994, and 1997, respectively. His major research interests spread parallel and distributed computing, system software, and programming languages. He is also a member of ACM, IEEE, and USENIX.