Phoenix: A Scalable Streaming Hypergraph Analysis Framework

Kuldeep Kurte, Neena Imam, S.M.Shamimul Hasan, and Ramakrishnan Kannan

Computing and Computational Sciences Directorate,
Oak Ridge National Laboratory **, Oak Ridge, TN, USA,
{kurtekr, imamn, hasans, kannanr}@ornl.gov

Abstract. We present Phoenix, a scalable hypergraph analytics framework for data analytics and knowledge discovery that was implemented on the leadership class computing platforms at Oak Ridge National Laboratory (ORNL). Our software framework comprises a distributed implementation of a streaming server architecture which acts as a gateway for various hypergraph generators/external sources to connect. Phoenix has the capability to utilize diverse hypergraph generators, including HyGen, a very large-scale hypergraph generator developed by ORNL. Phoenix incorporates specific algorithms for efficient data representation by exploiting hidden structures of the hypergraphs. Our experimental results demonstrate Phoenix's scalable and stable performance on massively parallel computing platforms. Phoenix's superior performance is due to the merging of high performance computing with data analytic.

Keywords: Hypergraph, scalable, streaming, high performance computing, graph clustering.

1 Introduction

Over the last few years, we have witnessed the explosive growth of data due to the technological advancements in the fields of social networking, e-commerce, smart mobile devices, etc. This necessitates the development of novel data mining/analysis approaches to address the various analytical challenges posed by the massive growth in data. Some examples of data analytics include live tracking in the transportation sector, fraud management in insurance, product recommendations in the retail industry, and predictive analysis in health care. These analyses study the relations, dynamics, and behavior at an individual-level (entity-level) as well as at the group-level. The graph representation, $G = (V, E)$, in which

** This manuscript has been authored in part by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

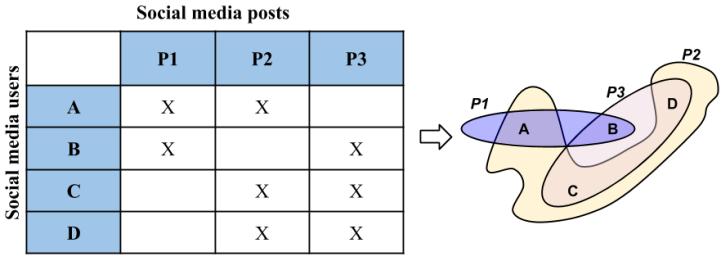


Fig. 1: Example hypergraph showing social media users (rows) and three social media posts (columns). Each post P_i is represented as an hyperedge and those users who interacted with that post are the hypergraph vertices incident on that hyperedge.

entities are represented by vertices ($V = \{v_1, v_2, .., v_n\}$) and relations among entities are represented by edges ($E = \{e_1, e_2, ..., e_m\}$), is a natural way to model such relational information. For instance, in an e-commerce system, customers and products are modeled as vertices and customer-product relations are represented by edges.

The graph representation of the information is able to capture the dyadic relations, i.e. relations between two entities but fails to model the group-level interactions. Due to the fact that the individual's behavior is mainly influenced by the group-level interactions, modeling group-level dynamics is important. Hypergraphs—the generalization of graphs; provides an excellent way to model the group-level interactions [5, 9, 28]. A hypergraph $HG = (V, H)$ is an ordered pair of 'n' vertices, i.e. $V = \{v_1, v_2, v_3, ..., v_n\}$, and H is a set of 'm' hyperedges, i.e. $H = \{H_1, H_2, H_3, ..., H_m\}$. Each hyperedge H_i is a vector of incident vertices such that $V \equiv h_1 \cup h_2 \cup h_3 \cup ... \cup h_m$. Figure 1 shows an example hypergraph which includes four social network users, A, B, C, D and three social media posts P_1, P_2, P_3. Each post P_i represents a hyperedge and its incident vertices are the users who interacted with the content, say, shared, liked, or commented on the post (represented by 'X'). From this example, it is evident that such hypergraph-based representation is useful to understand the information propagation among entities and the categorization of groups according to specific interests over the social network.

Although, the efficacy of hypergraphs for modeling group dynamics is well documented [1], efficient hypergraph analytics must overcome challenges associated with accurate hypergraph representation and scalable computation models that can deal with very high data ingestion rates without creating bottlenecks. While several large-scale graph processing software are available such as [18, 6, 26, 7], only a limited number of options are available for *hypergraph* analysis frameworks [28]. Very large scale hypergraph analysis requires scalable and distributed computing systems which present novel challenges as well as opportunities. The situation becomes more challenging when streaming data need to

be incorporated in the framework. Some challenges posed by the streaming scenario include, variability in the streaming rates from various external hypergraph sources, heterogeneity in representing the hypergraph, and efficient hypergraph representation at a system-level to sustain the streaming scenario.

Little research has been done for methodical performance evaluation of large-scale hypergraph analysis frameworks in a streaming scenario. The leadership class high performance computing facilities, such as hosted at Oak Ridge National Laboratory, provide petascale to exascale computing powers, large amounts of per node memory, efficient storage, and high-speed interconnects. Such leadership class computing facilities can meet the computational requirements of large-scale streaming hypergraph analysis. As such, researchers at Oak Ridge National Laboratory developed *Phoenix*, a high performance, hybrid system enabling concurrent utilization of online and offline analysis worlds. Phoenix architecture is distributed for scalability of problem size and performance. In addition, Phoenix is designed for fast and scalable ingest of streaming data sources. Phoenix also incorporates fast online (CRUD) operations and has dynamic (and fixed) schema. Using Phoenix, researchers are able to perform fast decoupled offline global analytics with in-memory snapshots and commit logs. Phoenix was deployed on Oak Ridge National Lab's Titan (ranked number one on top500[1] list in 2012) and showed good performance. Originally designed for simple graph analytics, we recently enhanced Phoenix to handle *hypergraph*s. The performance of Phoenix for streaming data sets is the subject of this paper.

In the following sections, we present our approach to scalable streaming hypergraph analysis as implemented in Phoenix. Section 2 presents an overview of the various hypergraph analysis tools. Section 3 presents the Phoenix framework for streaming hypergraph analysis and describes various technical aspects of Phoenix. Section 4 presents results of the numerical experiments we performed to evaluate metrics such as streaming performance, ingestion performance, and hypergraph clustering efficiency. Section 5 summarizes our observations and discusses few future extensions of this work.

2 Related Work

Many hypergraph analysis tools are available. However, none of these tools presents the scalability and flexibility associated with Phoenix. In addition, Phoenix incorporates scalable hypergraph generators. Most other hypergraph analytics software tools do not have this attribute. In the following paragraphs, we present an overview of the various hypergraph analysis tools and the advantages and disadvantages of each.

HyperNetX is a Python library that supports hypergraph creation, hypergraph-connected component computation, sub-hypergraph construction, hypergraph statistics computation (e.g., node degree distribution, edge size distribution, toplex size computation for hypergraphs), and hypergraph visualization (e.g.,

[1] https://www.top500.org/system/177975

draw hypergraphs, color nodes, and edges). *HyperNetX* was released in 2018 under the Battelle Memorial Institute license [21]. *HyperNetX* library does not support high performance computing (HPC) based parallel processing. Also, *HyperNetX* library documentation does not provide any scalability information.

Chapel HyperGraph Library (CHGL) was developed in the Chapel programming language by the Pacific Northwest National Laboratory. In the CHGL, users can use both shared and distributed memory systems for the storage of hypergraphs. The CHGL is not well documented and requires knowledge of the Chapel programming language, which is Partitioned Global Address Space (PGAS) language. PGAS languages are not as widely used as the C or C++ programming language. However, CHGL does offer valuable functionality within the context of parallel computations [4, 2].

HyperX offers a scalable framework for hypergraph processing and learning algorithms, which is developed on top of Apache Spark. It replicates the design model that is utilized within GraphX. *HyperX* directly processes the hypergraph rather than converting the hypergraph to a bipartite graph and employs GraphX to do the processing [15, 2]. Apache Spark programming paradigm cannot match the scalability offered by a leadership class computing platform.

HyperGraphLib package was developed in the C++ programming language, which supports k-uniform, k-regular, simple, linear, path search, and isomorphism algorithms. *HyperGraphLib* employs both OpenMP and Boost libraries. *HyperGraphLib* can not represent a hypergraph as a bipartite graph or a 2-section graph. Moreover, *HyperGraphLib* is not integrated with any graph libraries for advanced analytics [14, 2].

Halp is a Python library that provides both directed and undirected hypergraph implementations as well as a range of algorithms. These include a variety of hypergraph algorithmsfor instance, k-shortest hyperpaths as well as random walk and directed paths [13, 2]. However, *Halp* does not provide parallel implementation of the algorithms.

SAGE hypergraph generator was developed in the Python language and supports the creation of complete random, uniform, and binomial random uniform hypergraphs. Nevertheless, large scale hypergraph generation is not possible in *SAGE*. Besides, *SAGE* does not support parallel hypergraph generation.

Karlsruhe Hypergraph Partitioning (KaHyPar) was developed in C++ and is a multilevel hypergraph partitioning framework. It supports hypergraph partitioning with variable block weights and fixed vertices. Although *KaHyPar* is a useful tool, it does not support the hypergraph generation facility. [16, 24].

The Julia programming language was used to develop the *SimpleHypergraphs.jl* hypergraph analysis framework. It is an efficient hypergraph analysis tool that supports distributed computing. However, *SimpleHypergraphs.jl* is heavily dependent on the *HyperNetX* library, specifically for hypergraph visualization. Moreover, *SimpleHypergraphs.jl* tool provides limited hypergraph analysis functionalities and is not highly scalable [2].

networkR was developed in the R programming language, which supports hypergraphs projection into graphs. networkR also supports degree distribution,

diameter, centrality, and network density computation. One of the limitations of the *networkR* is that it needs to project hypergraph into graph structure for analysis. Moreover, vertices and hyperedge related meta-information is not available in networkR [20, 2].

Gspbox provides hypergraph modeling capability. Although in Gspbox, one can manipulate the hypergraph by transforming a model into a regular graph, it does not provide specific solutions or optimizations for hypergraphs [8, 2].

BalancedGo software was developed in the Go programming language. BalancedGo supports generalized hypertree decompositions via balanced separators. *BalancedGo* supports a limited number of algorithms mainly focused on hypertree decompositions. Moreover, *BalancedGo* supports only HyperBench format or PACE Challenge 2019 format [3] as input.

Pygraph was released under the MIT license and is a Python library that can be used to process graphs. It includes hypergraph support along with standard graph functionalities. However, Pygraph does not offer any hypergraph optimization feature [22, 2].

Yadati et al. developed *HyperGCN*, a new graph convolutional network (GCN) training approach for semi-supervised learning (SSL) on hypergraphs [30]. The Python implementation of the tool is available in [12]. The quality of the hypergraph approximation heavily depends on weight initialization, which is a limitation of *HyperGCN* [30].

Multihypergraph is a Python package that provides support for multi-edges, hyperedges, and looped edges. The main focus of the *Multihypergraph* package is the mathematical understanding of graph than algorithmic efficiency. Moreover, the *Multihypergraph* package is limited with graph model memory definition and isomorphism functionalities and does not provide any other functionalities for hypergraphs [19, 2].

d3-hypergraph is a hypergraph visualization tool developed on top of the D3 JavaScript library. Another example of the hypergraph visualization tool is *visualsc,* which is similar to the open-source graph visualization tool Graphviz. *d3-hypergraph* and *visualsc* tools are solely used for hypergraph visualization.

3 Framework for Analyzing Streaming Hypergraphs

This section describes the overall Phoenix framework and its various components which enable the analysis of the streaming hypergraph. Figure 2 shows Phoenix's end-to-end framework which is composed of various essential modules for analyzing the streaming hypergraphs in a distributed and scalable fashion.

3.1 Hypergraph Sources and Generation

Phoenix is capable of utilizing a diverse set of graph generators as inputs to the framework. One of the candidates is a distributed hypergraph generator called HyGen, which is capable of generating synthetic hypergraphs. HyGen is another high performance graph analytics project at Oak Ridge National Laboratory

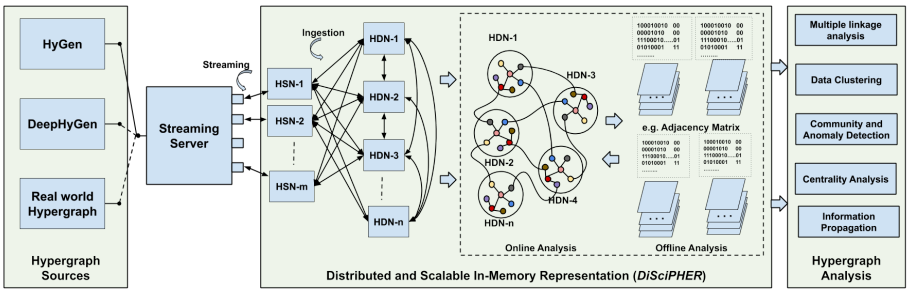


Fig. 2: Phoenix's end-to-end framework for scalable and distributed Hypergraph Analysis. Streaming server acts as a gateway where various hypergraph generators/external sources can connect. Next the streaming server streams the hypergraph in the form of hyperedge or incidences to the Graph Service Nodes (GSNs). GSNs handles the communication with the streaming server and consumes the hypergraph and sends it to the Graph Data Nodes (GDNs) where GDNs store the ingested hypergraph as its in-memory representation.

and was incorporated in the Phoenix architecture. HyGen takes input parameters such as number of clusters, number of vertices, and number of hyperedges to generate the corresponding hypergraph. For instance, if we have a rough understanding about the number of the clusters in the real-world hypergraph (e.g. communities), HyGen will enable the rapid production of the different sizes of hypergraphs which can be further consumed (by HSNs) and stored in-memory (by HDNs) in a distributed fashion. Refer Fig. 2 and sec. 3.3 for more information on HSNs and HDNs. Further, various online and offline analysis can be performed on this generated hypergraph. Similarly, the external hypergraph sources can also connect to the streaming server. More detailed discussions on graph generators can be found in references [32, 17, 27].

3.2 Hypergraph Streaming and Consumption

A streaming server is developed to facilitate the streaming of hypergraphs generated by hypergraph generators and from external sources to the internal core component called *DiSciPHER* (refer sec. 3.3) which is responsible for hypergraph consumption and in-memory storage. The three advantages of having this layer of streaming server are as follows:

1. **Decoupling**: Streaming server acts as a gateway and prevents hypergraph generators and external sources from directly accessing the *DiSciPHER* which is a core internal module of Phoenix. This provides the flexibility to make changes in the *DiSciPHER* module without impacting the accessibility of the hypergraph sources. Moreover, syntactic changes made by hypergraph sources do not have any impact on the *DiSciPHER*'s representation.

2. **Standardization**: Streaming servers can acquire data either as a bipartite representation or as a hyperedge representation. It is unlikely that all external sources comply with a unified syntax even though the data follow the semantics of bipartite or hyperedge representation. The streaming server can implement various methods for data translation to address this syntactic heterogeneity problem.
3. **Intermediate caching**: The rate of streaming from different external sources of hypergraphs can be different. At the system level, the heterogeneity in the streaming rates could cause data loss in case of extremely high data streaming and longer wait time for HSN processes in case of slow data streaming. We believe that the intermediate layer of the streaming server can stabilize the rate of streaming hypergraph from various external sources to HSN. The streaming server can provide a temporary storage capability to store the acquired hypergraph data before sending it to the HSNs of *DiSciPHER* module. This way streaming servers can stabilize the streaming rate.

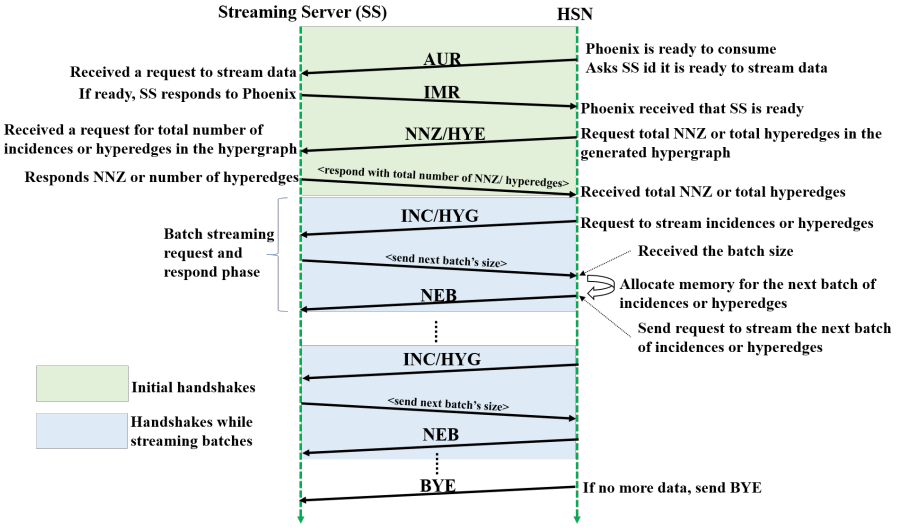


Fig. 3: Sequences of messages and data exchanges take place between the streaming server and hypergraph service node (HSN) process. The green portion depicts the message and data exchanges during the initial handshake. The blue portion depicts messages and data exchanges take place during streaming the batches of incidences (or hyperedges).

The streaming server can acquire hypergraphs in one of two ways: 1) Bipartite representation: a list of incidences and 2) Hyperedge representation: a list of hyperedges. Each incidence in a bipartite representation is a two dimensional vector

$\langle i, j \rangle$, such that $v_i \in H_j$, i.e. vertex v_i incident upon hyperedge H_j. On the other hand, the hyperedge representation constitutes a set of hyperedges (H) in which each hyperedge is a vector of incident vertices, i.e. $H_k = \langle v_{k1}, v_{k2}, v_{k3}, ..., v_{kp} \rangle$ and 'p' is the total number of incident vertices on hyperedge k.

The streaming server opens multiple communication ports where several Hypergraph Service Node (HSN) processes of *DiSciPHER* module, which is responsible for the consumption of the hypergraph, can connect and consume the hypergraph. In the case of bipartite representation, the streaming server performs streaming of incidences in a batched fashion. The batch size represents the maximum number of hypergraph incidences that can be packed in a batch. The batch size in case of hyperedge representation is the maximum number of hypergraphs per batch. Due to the variable size of hyperedges in a batch, the batch creation is not as straight-forward as in the bipartite representation. Here, each hyperedge is re-formatted as $\langle h_{id}, p, v_1, v_2, v_3, ..., v_p, -1 \rangle$ by appending hyperedge identifier h_{id}, its length in the beginning p, followed by a list of incident vertices, i.e. v_i and '-1' at the end to indicate the termination of the hyperedge. In this way, the hypergraphs are packed to form a batch such that each element in the batch represents either hypergraph identifier, length of hypergraph, vertex identifier, or '-1'.

As mentioned in the paragraph above, the hypergraph service node processes (HSNs) connect to the communication ports of the streaming server and consume a hypergraph either as a batched incidences or as hyperedges. We implemented a handshaking and communication protocol to enable the streaming and consumption of the hypergraphs. Figure 3 shows a sequence of commands and data exchanges occurring while streaming the hypergraph. Initial handshake includes HSN process sending a message "AUR" asking if the streaming server is ready to stream the hypergraph. HSN waits until it receives "IMR" from the streaming server which indicates that the server is ready to stream the hypergraph. Next, depending on the format in which HSN wants to consume the hypergraph, it either sends "NNZ" to ask for the number of incidences (non-zeros) in case of bipartite representation or sends "HYG" to ask for the number of hyperedges in the hypergraph. In response to this message, streaming server sends total number of incidences (non-zeros) or total number of hyperedges to HSN.

After this initial handshake (as depicted in the green portion of Fig. 3), the streaming server sends "INC" (for bipartite) or "HYG" (for hyperedge) message to the streaming server. After receiving this message, the streaming server prepares the batch of incidences (or hyperedges) and sends the batch size to the HSN so that HSN can reserve sufficient memory to consume the upcoming batch of incidences (or hyperedges). Further, HSN sends a"NEB" message to the streaming server to indicate that it is ready to consume the batch of incidences (or hyperedges). Upon receiving "NEB", the streaming server sends the prepared batch of incidences (or hyperedges) to HSN. This communication between HSN and the streaming server (as depicted by the blue portion in Figure 3) continues until the entire hypergraph is consumed by HSN.

3.3 Distributed and Scalable in-memory rePresentation of HyERgraph (DiSciPHER)

Phoenix's *DiSciPHER* module is responsible for the efficient in-memory representation of the consumed hypergraph such that it enables both offline and online analysis on the streaming hypergraphs. Here, we describe, 1) how the *DiSciPHER* module represents hypergraphs and 2) two essential components of *DiSciPHER* which enable the efficient in-memory representation, i.e. Hypergraph Service Node (HSN) processes and Hypergraph Data Node (HDN) processes.

In-Memory Representation of Hypergraph In this subsection, we describe the in-memory representation of hypergraphs at a system level which enables the online and offline analysis of hypergraphs in a streaming scenario inside Phoenix framework. In a streaming scenario, it is highly likely that only the part of hyperedge is available which is being streamed at any given time. In other words, the complete knowledge of the incident vertices of an hyperedge is not available at the time when that hyperedge is being streamed. The remaining (partial) hyperedge can arrive later. This characteristic of the streaming scenario, along with the variable sized nature of the hyperedge, increases overhead of the hypergraph ingestion process. For instance, every time the partial hyperedge arrives, the hyperedge insertion involves updating and re-distributing the vertices (or hypergraphs) among the compute nodes (i.e. HDNs in Phoenix).

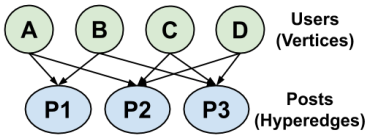


Fig. 4: Bipartite representation used in Phoenix to represent hypergraph. Example hypergraph is represented in the Fig. 1. Each social media post is an hyperedge and users who share, like or comment on a common post, are the vertices incident upon that hyperedge. In bipartite representation, both vertices and hyperedges are represented by nodes and additional edges are added from incident vertices to their hyperedge to preserves the semantics of the hyperedge.

To alleviate the hyperedge ingestion overhead problem in a streaming scenario, we adopted a bipartite representation to represent the hypergraph at a system level. Figure 4 shows the bipartite representation of the hypergraph shown in the Fig. 1. In this representation (refer Fig. 4), users are vertices represented as green circles on the top and each social media post is a hyperedge shown as a blue circle at the bottom. The additional edges (E_{vh}) are inserted from the nodes on the top side (vertices) to the hyperedge nodes on the bottom

side to represent the incident vertices of the hyperedge. This approach translates to hyperedge insertions with multiple edges. Such a representation in streaming scenarios can accommodate hyperedge with partial information, where the complete vertex set is still unknown. However, this approach increases the number of edge insertions for a given hyperedge. So there is a trade-off between update operations to accommodate the partial hyperedge issue and increased number of edge insertions in case of bipartite representation. In our opinion, the scalable and distributed nature of Phoenix can handle the increased number of edge insertions without impacting the streaming performance.

Hypergraph Service Nodes (HSNs) and Hypergraph Data Nodes (HDNs) *DiSciPHER* is an essential component of the Phoenix's ecosystem consisting of hypergraph service nodes (HSNs) and back-end data storage and processing nodes (HDNs). The responsibilities of HSNs include communicating with the streaming server, consuming the hypergraph, redirecting the consumed hypergraph to the HDNs in a load-balanced fashion, and keeping track of the progress of the system by broadcasting messages to HSNs and HDNs. HDN's main task is to process and store the consumed hypergraph in a distributed in-memory fashion.

DiSciPHER's implementation is rather memory intensive (i.e. best suited for large clusters and supercomputers with very large memory). *DiSciPHER*'s deployment typically contains a number of HSNs (service nodes) that could be load balanced, and a larger number of HDNs (data nodes)—depending on the size of the graph to be processed. A balanced shared distributed memory – multi-processing and multi-threading design approaches are used to achieve the best concurrent performance.

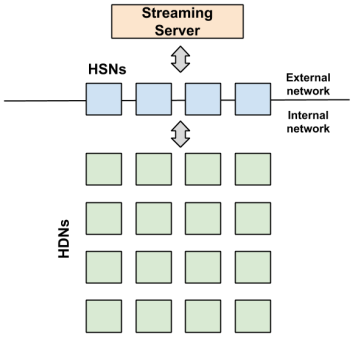


Fig. 5: Cluster view of HSNs (blue boxes) and HDNs (green boxes).

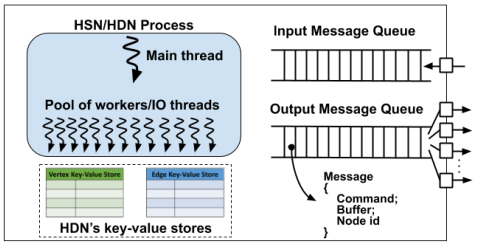


Fig. 6: Programmer's view of Hypergraph Service and Data Nodes.

Figure 5 shows the cluster view of the HSNs and HDNs inside *DiSciPHER*. The number of HSNs and HDNs to be used (or deployed) depends on the size of the hypergraph and the availability of the compute resources, i.e. size of the cluster and memory per compute node. The *DiSciPHER* module makes sure that the HSNs and HDNs are deployed before initiating the communication. The process begins by allocating the sufficient numbers of compute nodes. Once the sufficient nodes are allocated, they are divided into two groups. One set of nodes are grouped as HSNs and remaining as HDNs. We designed this cluster as a multi-processing and multi-threaded architecture in which each node has a main process and several worker and I/O threads.

Figure 6 shows the programmers view of the main components of HSNs and HDNs. The master process of HSN-'0' collects the network addresses of all the nodes in the cluster and groups them as HSNs and HDNs. Each node has dedicated queues for input and output messages. The input message queue is connected to the input socket which is setup to receive the incoming messages from all other nodes. Similarly, multiple output sockets are set up, each is dedicated to one node in the cluster. The output message queue is connected with these output sockets. The message (incoming or outgoing) contains *command*: indicates task to be performed, *Buffer*: contains data to operate on and *Node id*: identifier of the destination HSN or HDN node. The messaging scheme supports peer-to-peer and broadcasting of the messages. The HDNs have additional in-memory distributed local key-value storage to store the local vertices (or hypergraph node) and edges (connecting vertex and hypergraph node), and their mapping with the corresponding global identifiers.

The input and output message queues are thread-safe and accessed by both workers and I/O threads within the node. The I/O threads are responsible for dispatching the outgoing messages, queued in the output message queue, to the appropriate socket based on the destination node. The I/O threads are also responsible for receiving the incoming messages from the input socket and putting them in the input message queue. The worker threads, depending on the requirements, form messages and put them in the output message queue and retrieve messages from input message queue and further perform the required task.

Once the cluster of HSNs and HDNs is deployed and ready to consume the hypergraph, HSN connects to the streaming server and requests for a batch of either incidences or hyperedges. Since, at the system level, *DiSciPHER* represents hypergraph in a bipartite form, the HSN creates message for inserting edge, i.e. E_{vh} (representing incidences connecting hypergraph's vertex v and hyperedge, h and directs it to the appropriate HDN based on the source vertex identifier. The HSN balances the load among HDNs by distributing the E_{vh} to HDNs in a round-robin fashion based on the identifier of the source vertex v.

The message for inserting E_{vh} includes *command*: inserting edge (E_{vh}), *buffer*: containing *source id* representing hypergraph vertex identifier, *destination id* representing hyperedge identifier, and *edge id* an identifier representing E_{vh}, and *node id* representing the identifier of destination HDN. The worker threads of HSN are responsible for creating these messages and putting them

in the output message queues. The I/O threads periodically take these messages out of the output message queue and send them to the appropriate HDNs through their dedicated communication sockets. The I/O threads of HDNs receive the input massages from the receiver socket and put them in the input message queue. The worker threads of HDNs further remove the messages from input queue and perform the required tasks, in this case storing the vertex (v) and the edge connecting them, i.e. E_{vh}.

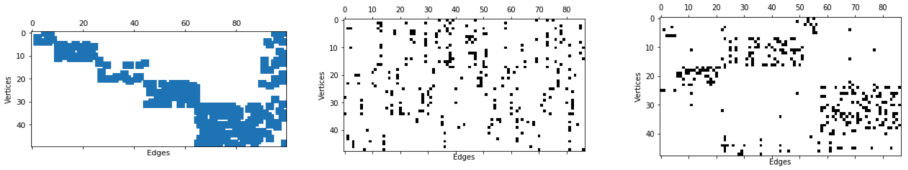
If the destination vertex representing the hyperedge identifier belongs to the different HDN then the worker thread of the current HDN forwards the edge insertion message to the appropriate HDN based on the hyperedge identifier. In this case, both the HDNs store the edge E_{vh} such that, the destination field of the edge in the first HDN's points to the second HDN's identifier and the source field of the edge in the second HDN points to the identifier of the first HDN. The current implementation of Phoenix requires hypergraph generators (and external sources of hypergraph) to differentiate the vertex and hyperedge identifiers, in order to avoid potential conflicts between the vertex and hyperedge.

3.4 Hypergraph Clustering

The graph clustering problem involves partitioning the vertices, such that the similarity of vertices within a cluster is higher than the inter-cluster similarity. While most approaches on graph clustering assume edges as pairwise relationships between vertices, many real world applications participate in multi-way relations represented as hyperedges in hypergraphs. Analogous to the graph clustering task, hypergraph clustering seeks to find partitions among vertices using hyperedges[25].

Within the ML community, the seminal work of Zhou, Huang, and Schölkopf [33] looked at learning on hypergraphs. They sought to support Spectral Clustering methods on hypergraphs and defined a suitable hypergraph Laplacian. This effort, like many other existing methods for hypergraph learning, makes use of a reduction of the hypergraph to a graph [1]. We apply similar techniques on this paper.

Formally, in this paper, given a hypergraph $HG = (V, H)$, we determine k partitions on V, $\pi_V = \pi_1, \pi_2, \cdots, \pi_k$, where $\pi_i \subset V$, $\cup_i \pi_i = V$ and $\cap_i \pi_i = \emptyset$.



a Ground Truth b Input Graph c Hypergraph Clustering

Fig. 7: Hypergraph Clustering Demonstration

Let us consider, there are $|V| = m$ vertices in the hypergraph and $|E| = n$ hyperedges of the hypergraph is represented as a $P \in \mathbb{R}^{m \times n}$. We can perform the spectral cut on P as follows.

Say, $D_v \in \mathbb{R}^{m \times m}$ and $D_e \in \mathbb{R}^{n \times n}$ be diagonal matrices of row and column sums of H. Determining the top-k eigenvectors of laplacian matrix $S = D_v^{-\frac{1}{2}} H D_e^{-1} H^T D_v^{-\frac{1}{2}}$ will provide the soft clustering on V. Obtaining the highest cluster membership of each v_i will provide us k clusters of the hypergraph clustering π_V.

In our case S is a sparse symmetric case and we are determining k leading eigen vector as value decomposition $S = UDU^T$, where $U \in \mathbb{R}^{n \times k}$. Algorithm 1 presents the listing of the high performance spectral clustering for hypergraphs.

Algorithm 1: Hypergraph Clustering

Input: $H \in \mathbb{R}^{m \times n}$, k clusters
Output: Vertex cluster π_V
1 Compute $D_v = row\_sum(H)$;
2 Compute $D_e = column\_sum(H)$;
3 Compute $S = D_v^{-\frac{1}{2}} H D_e^{-1} H^T D_v^{-\frac{1}{2}}$;
`/* eigsh is eigen value decomposition for symmetric square matrix. */`
4 ;
5 Compute eigen vectors $U = eigsh(S, k)$;
6 Compute $\pi_V = \text{argmax } U$

In this paper, we used Scalable Library for Eigenvalue Problem Computations (SLEPC) [10] for computing the eigen value decomposition problem in Step 5 of the above algorithm for scaling to very large hypergraphs in distributed MPI environment.

The output of the above algorithm for a generated hypergraph HG is shown in Figure 7. We generated a ground truth graph as shown in Figure 7a and permuted the rows and columns as in Figure 7b. We took this sparse random hypergraph HG and determined five clusters. The output is shown in Figure 7c.

4 Performance

In this section, we describe the performance evaluation of the various components of the Phoenix framework. We first discuss the dataset, computation environment, software environment, performance metrics considered for this performance evaluation process and further present the performance results.

4.1 Approach for Evaluating the Performance

As a proof-of-concept, we evaluated the performance of various components of the Phoenix framework for streaming hypergraphs. Here, we specifically focused on the hypergraph ingestion performance. We evaluated the streaming performance with varying batch sizes. We also investigated ingestion performance with

varying numbers of HSNs and HDNs. Finally, we evaluated the performance of the distributed hypergraph clustering approach.

Dataset We used the synthetic hypergraphs generated by our distributed hypergraph generator, i.e. HyGen, by varying the parameters such as #clusters, #vertices and #hyperedges. Table 2 and 3 show various synthetic hypergraphs (generated using HyGen) used for this performance evaluation.

Computational Environment We used Oak Ridge Leadership Computing Facility (OLCF) called Rhea-cluster. It is a 521-node commodity-type Linux® cluster. Each node of Rhea contains two 8-core 2.0 GHz Intel Xeon processors with Intel's Hyper-Threading (HT) Technology and 128GB of main memory. Rhea also has nine GPU nodes and each node is equipped with 1TB of main memory and two NVIDIA K80 GPUs with two 14-core 2.30 GHz Intel Xeon processors with HT Technology. Rhea is connected to the OLCF's high performance Lustre® filesystem, Atlas, through a high-speed interconnect 4X FDR Infiniband with maximum data transfer rate of 56GB/s. More information on the specification of Rhea can be found at [23].

Software Environment The codebase of Phoenix is developed in C++ (specifically C++11 standards). Inter-node communication is implemented using binary message structures over a message-oriented middleware. Currently ZeroMQ over Transmission Control Protocol (TCP) is implemented [11], and MPI is planned for the future developments. ZeroMQ is a high-performance asynchronous messaging library that supports common messaging patterns (pub/sub, request/reply, client/server and others) over a variety of transports (TCP, in-process, inter-process, multicast, WebSocket and more). Intel®'s Thread building blocks, version 4.3+ is used to develop a scalable implementation of the concurrent queues. Further, we used the Scalable Library for Eigenvalue Problem Computations (SLEPC) [10] for computing the eigenvalue decomposition.

Performance Metrics We mainly measured the performance in terms of streaming rate in a batched streaming scenario, ingestion rate with different settings, and scaling performance of the developed hypergraph clustering method.

4.2 Results

First, we present the time performance of the streaming hypergraph from streaming server to HSN of *DiSciPHER*. Although the additional layer of a streaming server provides few architectural benefits (refer Section 3.2), this experiment is necessary to understand its overall overhead. Once the hypergraph data is acquired at streaming server from hypergraph generators and external sources, the streaming server further streams the data in the batches of incidences (or hyperedges) instead of streaming only one incidence at a time (refer Section 3.2 for

streaming strategies). The overhead includes batch preparation time followed by the time to stream those batches. Table 1 presents the total batch preparation and streaming timings for different sizes of batches. One more motivation behind this experiment was to understand the ideal batch size for streaming hypergraph data. From the timings shown in the Table 1, it is clear that, although the batch size is varying, the overall batch preparation and streaming timings are roughly same for all the scenarios, i.e. ≈8.2 sec. and ≈2.1 sec. for batch preparation and streaming respectively.

As mentioned in the Section 3.2, the external sources of hypergraph can stream data in varying rates and formats. At the system level, such heterogeneity in the streaming rates could cause the loss of data in case of extremely high data streaming rates and longer wait times for HSN processes in case of slow data streaming rates. Based on the results of the batched streaming experiments in the Table 1, we argue that the intermediate layer of the streaming server can stabilize the rate of streaming hypergraph from various external sources to HSN.

Next, we describe the time performance of the scaling experiments in two different scenarios: 1) weak-scaling, i.e. increasing the number of incidences (increasing hypergraph size) with number of HDNs and 2) strong-scaling, i.e. adding more HDNs for a fixed number of incidences (i.e. fixed sized hypergraph). We want to mention that, in both settings one compute node was used as HSN; each HSN has 12 worker threads and two I/O threads. Each HDN has eight worker threads and two I/O threads. As described earlier, HDNs are responsible for storing hypergraphs in memory and perform necessary computations for its consumption. We carried out both weak and strong scaling experiments in two different settings. In the first setting, various hypergraphs were generated in which #vertices<#hyperedges and in the second setting various hypergraphs were generated in which #vertices>#hyperedges. The motivation behind these experiments was to analyze the ingestion performance for the streaming data on a leadership class computing platform.

In the weak-scaling experiment, one compute node was used as HSN and the number of compute nodes used for HDNs were increased along with the hypergraph size. Table 2 shows two different settings which were used to generate hypergraphs for weak-scaling experiments. #Incidences indicate the size of the hypergraph. Ideally, a constant ingestion time is expected in this weak-scaling experiment as the workload of hypergraph consumption per HDN roughly re-

Table 1: Timing for batched streaming of hypergraph data from streaming server to HSN. ≈2.5M hyperedges and ≈208M incidences (NNZ) used.

Streaming batch size (#nbatches)	**Total batch preparation overhead (sec.)**	**Total Streaming time (sec., excluding batch preparation)**
300M (1 batch)	8.265	2.108
200M (2 batches)	8.268	2.08
100M (3 batches)	8.265	2.097
50M (5 batches)	8.255	2.086
10M (21 batches)	8.271	2.126

mains the same with the growing size of hypergraph and number of HDNs. In our case, one HSN is used and the total ingestion time includes the time that HSN takes to prepare messages and send it to the respective HDNs. For the same reason, as the hypergraph size increases, we expect some linear growth in the HSN's contribution to the total ingestion time, however, in an ideal scenario, the HDNs consumption timing should be constant. Along with the ingestion time, we also measured the ingestion rate, i.e. number of incidences ingested per second. In an ideal scenario, the ingestion rate should grow as we increase the hypergraph size and number of HDNs.

Except the first column in the Table 2, the other columns represent different scenarios. The main goal of this experiment is to understand the variations in the ingestion times and ingestion rates while increasing the hypergraph size and number of HDNs. We measured total ingestion time and derived the ingestion rates. Table 2 shows that the total ingestion time is increasing with increasing hypergraph size and HDNs. Figure 8 shows the variations in the ingestion timings for different scenarios for both settings. It can be seen that the ingestion rate increased for the first three scenarios and after that it remained stable, i.e. ≈3.3M ingestion per second. The potential reasons for the increase in the ingestion time and the ingestion rate are use of single HSN and background network traffic created by other jobs executing on Rhea cluster. However, we would like to emphasize the fact that we observed the stable ingestion performance in both the settings, i.e. one with #vertices<#hyperedges and other with #vertices>#hyperedges which represent to different hypergraph structures.

In the strong-scaling experiment, we kept the hypergraph size the same and increased the number of HDNs used for consumption. The intent behind this experiment is to understand the workload sharing ability of the HDNs when the

Table 2: Setting for weak-scaling and ingestion timings.

Weak scaling1 (#Vertices <#Hyperedges)					
#Clusters	1000	3000	6000	12,000	24,000
#Vertices	60,000	200,000	400,000	800,000	1,600,000
#Hyperedges	200,000	600,000	1,200,000	2,400,000	4,800,000
#Incidences	2,388,362	16,042,887	56,189,594	208,488,077	786,869,455
#HDN	1	2	4	16	64
Ingestion Time (Sec.)	3.5	6.6	17.1	63.1	248.0
Ingestion rate (#ing/sec.)	≈682K	≈2.4M	≈3.3M	≈3.3M	≈3.2M
Weak scaling2 (#Vertices >#Hyperedges)					
#Clusters	300	1000	2000	4000	8000
#Vertices	200,000	600,000	1,200,000	2,400,000	4,800,000
#Hyperedges	60000	200,000	400,000	800,000	1,600,000
#Incidences	5,437,372	23,855,054	71,276,112	239,667,974	859,300,720
#HDN	1	2	4	16	64
Ingestion Time (Sec.)	4.7	8.7	21.7	72.2	269.9
Ingestion rate (#ing/sec.)	≈1.1M	≈2.7M	≈3.3M	≈3.3M	≈3.2M

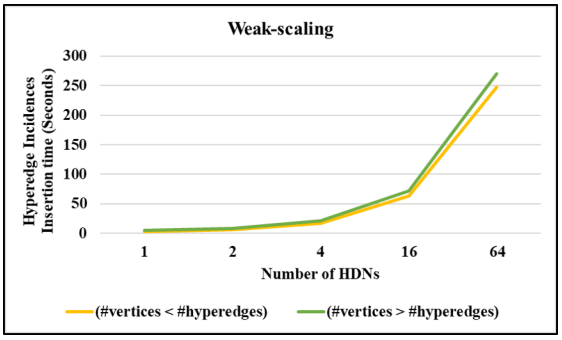


Fig. 8: Ingestion time for different scenarios for both the weak-scaling settings with varying numbers of HDNs. The ingestion time/rate and overall scaling of the hypergraph ingestion in a streaming scenario is largely determined by the number HSNs.

hypergraph is fixed and we add more HDNs. We measured the total ingestion time for these scenarios and derived the ingestion rate (refer Table 3). Table 3 shows the ingestion timings and rates for two strong-scaling settings, each with different number of NNZ (incidences) with increasing number of HDNs and Figure 9 shows the ingestion timings for two strong-scaling scenarios. Ideally, we expect a decreasing trend in the ingestion time as workload of hypergraph consumption per HDN decreases with the increase in the number of HDNs. From 3 and Figure 9, we can observe that the total ingestion time for the strong-scaling settings decreased when two HDNs were used, however, the ingestion time remained roughly constant for all of the subsequent scenarios. Similarly, in ideal cases, the ingestion rate in a strong-scaling setting should increase with the addition of more HDNs due to the decrease in the ingestion time for the constant workload. The ingestion rate showed some increase for the first two scenarios but remained nearly constant for other settings where the number of HDNs are increasing. We emphasize that the role of the HSN is to formulate messages and distribute the hypergraph data to the HDNs. The HDNs are responsible for the necessary computation and communications with other HDNs to store the consumed hypergraph in memory. The potential reasons for the deviation from the ideal strong-scaling behavior could be attributed to the use of a single HSN resulting and the increased message communication among HDNs with the increase in the number of HDNs.

As mentioned above, in the strong-scaling scenario, one can expect an increased ingestion rate with increase in the number of HDNs when the hypergraph size is kept constant, however, the results show some deviation from this ideal behavior. It should be noted that both strong and weak scaling experiments were performed with one HSN only which could be a potential reason for this performance deviation. Therefore, to understand the impact of varying number of HSNs, we fixed the number of HDNs to 512 and varied the numbers

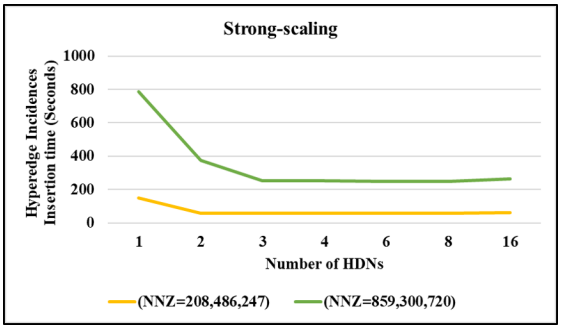


Fig. 9: Ingestion time for different scenarios for both the strong-scaling settings with fixed hypergraph size and varying number of HDNs. The ingestion time/rate and overall scaling of the hypergraph ingestion in a streaming scenario is largely determined by the number HSNs.

Table 3: Settings for strong-scaling experiment and ingestion timing.

	#HDN	Ingestion Time (Sec.)	Ingestion rate (#ing./sec.)
Strong scaling1 #clusters:12,000; #vertices:800,000; #hyperedges:2,400,000 NNZ=208,486,247	1	151.3	1.3M
	2	58.6	3.5M
	3	58.1	3.5M
	4	58.7	3.5M
	6	58.4	3.5M
	8	58.6	3.5M
	16	63.1	3.3M
Strong scaling2 #clusters:8,000; #vertices:4,800,000; #hyperedges:1,600,000 NNZ=859,300,720	1	786.7	1.1M
	2	373.2	2.3M
	3	251.6	3.4M
	4	252.8	3.4M
	6	250.6	3.4M
	8	248.6	3.5M
	16	263.9	3.3M

of HSNs from 1 to 8. Figure 10 shows the variation in the ingestion timing with the increasing number of HSNs. The results are favorable and it can be clearly observed that the ingestion time decreases with increase in the number of HSNs for a fixed problem size. Therefore we expect to see improved weak and scaling experiments by increasing the number of HSNs. Further analysis to understand the optimal number of HSNs is one of the objectives of our future research. In future, we intend to perform similar scaling experiments on even larger scale systems such as Oak Ridge National Laboratory's Summit supercomputer, which currently holds the number 1 spot on the top500[2] list [29].

Figure 11 shows the strong-scaling performance of the distributed hypergraph clustering algorithm (refer Section 3.4). In an ideal setting for strong-scaling, the total execution timing of hypergraph clustering and an average MPI message length should be decreased with the increasing number of MPI processes. We

[2] https://www.top500.org/system/179397

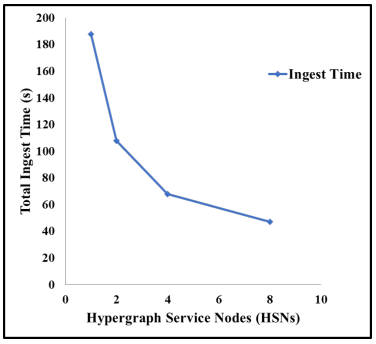


Fig. 10: Ingestion time variation with increase in HSNs with fixed 512 HDNs. The hypergraph ingestion rate increases significantly by increasing HSNs.

observed that, both, the job execution time and message length decreased exponentially with the increase in the MPI processes. The results showed an $\approx 38\times$ speedup when 64 MPI processes were used for hypergraph clustering.

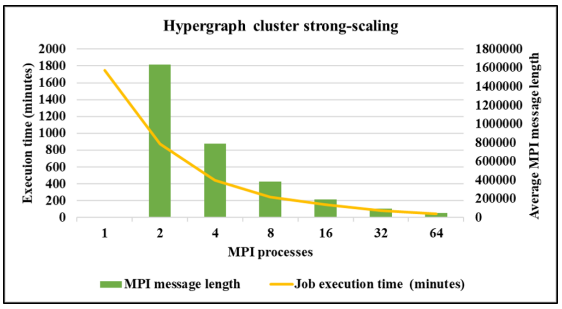


Fig. 11: Strong-scaling performance of the distributed hypergraph clustering analysis algorithm. Observed 38speedup when 64 MPI processes were used.

4.3 Observations

From the experiments performed to analyze the performance of the Phoenix framework we draw following key observations which should inform future research and development in hypergraph analysis:

- The additional layer of the streaming server is important in the streaming hypergraph scenario to stabilize the streaming process.
- The ingestion time/rate and overall scaling of the hypergraph ingestion in a streaming scenario is largely determined by the number HSNs.

- The hypergraph ingestion rate increases significantly with the increasing number of HSNs. However, further and performance analysis is required to obtain the optimal number of HSNs for a given size of the hypergraph.
- The distributed hypergraph clustering algorithm showed $\approx 38\times$ speedup when 64 MPI processes were used. More experiments are needed with larger hypergraphs to further validate the usefulness of the algorithm.

5 Conclusion

Graphs are becoming ubiquitous and growing in volume. From social networks to language modeling, the growing scale and importance of graph data have driven the development of numerous graph analytic systems. While graph analytic systems have many applications, they are not able to model group-level interactions with high fidelity. In this paper, we present our approach to hypergraph analysis to better capture the nuances of complex multilateral relations in group interactions. Although other hypergraphh analytic tools exist, they are not well suited to tasks such as generating the hypergraphs, modifying hypergraph structures, or expressing computation that spans multiple graphs and compute nodes.

In this paper, we present Phoenix, a scalable hypergraph analytics framework that was implemented on the leadership class computing platforms at Oak Ridge National Laboratory. Our software framework is implemented in a distributed fashion. Phoenix has the capability to utilize diverse hypergraph generators, including HyGen, a very large scale hypergraph generator developed by Oak Ridge National Laboratory. Phoenix also incorporates specific algorithms for efficient data representation by exploiting hidden structures of the hypergraphs. We presented experimental results that demonstrate Phoenix's scalable and stable performance on massively parallel computing platforms. In the future, we will optimize our load balancing techniques for better strong and weak scaling performances. Also, we plan to implement 2-D partitioning techniques to improve the scalability of HyGen [31]. Other future directions include the development of machine learning-based hypergraph generators, which will learn structures of real-world hypergraphs, and based on that information, the hypergraph generator will generate massive-scale hypergraphs.

Acknowledgements

Support for this work was provided by the United States Department of Defense. We used resources of the Computational Research and Development Programs and the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] Sameer Agarwal, Kristin Branson, and Serge Belongie. "Higher order learning with graphs". In: *Proceedings of the 23rd international conference on Machine learning*. 2006, pp. 17–24.

[2] Alessia Antelmi et al. "SimpleHypergraphs.jl–novel software framework for modelling and analysis of hypergraphs". In: *Inter. Workshop on Algorithms and Models for the Web-Graph*. Springer. 2019, pp. 115–129.

[3] *BalancedGo*. `https://github.com/cem-okulmus/BalancedGo`. [Online; accessed 2020-04-03]. 2020.

[4] *Chapel Hypergraph Library*. `https://github.com/pnnl/chgl`. [Online; accessed 2020-04-03]. 2020.

[5] Estrada E and Rodriguez-Velazquez J. "Complex networks as hypergraphs". In: *Arxiv preprint physics* 0505137 (2005).

[6] D. Ediger et al. "STINGER: High performance data structure for streaming graphs". In: *2012 IEEE Conference on High Performance Extreme Computing*. 2012, pp. 1–5.

[7] G. Feng, X. Meng, and K. Ammar. "DISTINGER: A distributed graph data structure for massive dynamic graph processing". In: *2015 IEEE International Conference on Big Data (Big Data)*. 2015, pp. 1814–1822.

[8] *Graph Signal Processing Toolbox (GSPBox)*. `https://github.com/epfl-lts2/gspbox`. [Online; accessed 2020-04-03]. 2020.

[9] Benjamin Heintz and Abhishek Chandra. "Beyond Graphs: Toward Scalable Hypergraph Analysis Systems". In: *SIGMETRICS Perform. Eval. Rev.* 41.4 (Apr. 2014), pp. 94–97. ISSN: 0163-5999. DOI: `10.1145/2627534.2627563`. URL: `https://doi.org/10.1145/2627534.2627563`.

[10] Vicente Hernandez, Jose E Roman, and Vicente Vidal. "SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems". In: *ACM Transactions on Mathematical Software (TOMS)* 31.3 (2005), pp. 351–362.

[11] Pieter Hintjens. *ZeroMQ: messaging for many applications*. " O'Reilly Media, Inc.", 2013.

[12] *HyperGCN: A New Method of Training Graph Convolutional Networks on Hypergraphs*. `https://github.com/malllabiisc/HyperGCN`. [Online; accessed 2020-04-03]. 2020.

[13] *Hypergraph Algorithms Package*. `https://murali-group.github.io/halp/`. [Online; accessed 2020-04-03]. 2020.

[14] *HyperGraphLib*. `https://github.com/alex-87/HyperGraphLib`. [Online; accessed 2020-04-03]. 2020.

[15] W. Jiang et al. "HyperX: A Scalable Hypergraph Framework". In: *IEEE Transactions on Knowledge and Data Engineering* 31.5 (2019), pp. 909–922.

[16] *KaHyPar - Karlsruhe Hypergraph Partitioning*. `https://github.com/kahypar/kahypar`. [Online; accessed 2020-04-03]. 2020.

[17] Jure Leskovec et al. "Kronecker graphs: An approach to modeling networks". In: *Journal of Machine Learning Research* 11.Feb (2010), pp. 985–1042.

[18] A. Lugowski et al. “Scalable complex graph analysis with the knowledge discovery toolbox”. In: *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2012, pp. 5345–5348.

[19] *multihypergraph*. `https://github.com/vaibhavkarve/multihypergraph`. [Online; accessed 2020-04-03]. 2020.

[20] *networkR - An R package for analysing social and economic networks*. `https://github.com/O1sims/networkR`. [Online; accessed 2020-04-03]. 2020.

[21] *pnnl/HyperNetX*. `https://pnnl.github.io/HyperNetX/build/index.html`. [Online; accessed 2020-04-03].

[22] *Pygraph*. `https://github.com/jciskey/pygraph`. [Online; accessed 2020-04-03]. 2020.

[23] *RHEA: A conduit for large-scale scientific discovery by pre- and post-processing and analysis of simulation data*. url=https://www.olcf.ornl.gov/olcf-resources/compute-systems/rhea/. (accessed on April 14, 2020).

[24] Sebastian Schlag et al. “K-way hypergraph partitioning via n-level recursive bisection”. In: *2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2016, pp. 53–67.

[25] Prithviraj Sen et al. “Collective classification in network data”. In: *AI magazine* 29.3 (2008), pp. 93–93.

[26] Narayanan Sundaram et al. “GraphMat: High Performance Graph Analytics Made Productive”. In: *Proc. VLDB Endow.* 8.11 (July 2015), pp. 1214–1225. ISSN: 2150-8097. DOI: 10.14778/2809974.2809983. URL: https://doi.org/10.14778/2809974.2809983.

[27] Jared Winick and Sugih Jamin. *Inet-3.0: Internet topology generator*. Tech. rep. Technical Report CSE-TR-456-02, University of Michigan, 2002.

[28] M. M. Wolf, A. M. Klinvex, and D. M. Dunlavy. “Advantages to modeling relational data using hypergraphs versus graphs”. In: *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 2016, pp. 1–7.

[29] D. E. Womble et al. “Early experiences on Summit: Data analytics and AI applications”. In: *IBM Journal of Research and Development* 63.6 (2019), 2:1–2:9.

[30] Naganand Yadati et al. “HyperGCN: A New Method For Training Graph Convolutional Networks on Hypergraphs”. In: *Advances in Neural Information Processing Systems*. 2019, pp. 1509–1520.

[31] Andy Yoo, Allison H Baker, and Roger Pearce. “A scalable eigensolver for large scale-free graphs using 2D graph partitioning”. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 2011, pp. 1–11.

[32] Jiaxuan You et al. “Graphrnn: Generating realistic graphs with deep auto-regressive models”. In: *arXiv preprint arXiv:1802.08773* (2018).

[33] Dengyong Zhou, Jiayuan Huang, and Bernhard Schölkopf. “Learning with hypergraphs: Clustering, classification, and embedding”. In: *Advances in neural information processing systems*. 2007, pp. 1601–1608.