Date: **June 05, 2013**	**High Level Design** **Big Data-HPC Bridge : The ACG Ingress** **FOR EXTREME-SCALE COMPUTING RESEARCH AND DEVELOPMENT (FAST FORWARD) STORAGE AND I/O**

High Level Design

LLNS Subcontract No.	B599860
Subcontractor Name	Intel Federal LLC
Subcontractor Address	2200 Mission College Blvd. Santa Clara, CA 95052

Table of Contents

Revision History

Date	Revision	Author
02/26/2013	1.0	Arnab Paul, Jaewook Yu, Kyle Ambert
03/04/2013	1.1	Arnab Paul, Jaewook Yu, Kyle Ambert
05/29/2013	1.2	Arnab Paul, Jaewook Yu, Kyle Ambert
06/05/2013	1.3	Arnab Paul

Changes (06-05) from the earlier version (03/04):

- **New Sections: 4.2, 4.4, 4.5**
- **Slightly Modified Sections: 4.6 (added a few more specifics on the dataset), and 5.**

1 Introduction

The scope of this document is limited to the design for the first set of deliverables related to the ACG solution architecture, i.e., generation and storage of big data graphs.

The ACG ingress pipeline starts with raw data at the ingress that must be pre-processed to extract graph structures and associated network information. To be used in the HPC world, these structures (graph and network information) will be represented in HDF5 format. The ingress will partition these structures appropriately, each partition or sub-partition small enough to fit in memory, so that the partitions can be distributed over HPC compute nodes for processing. The partitions can be split further whenever necessary in sliding windows slices (small enough to fit in-core).

The key component in establishing the Big Data-HPC bridge is an HDF5 adaptation layer (HAL). The HAL lays out arbitrarily connected graphs on a parallel or distributed storage system in HDF5 data-format, and acts as the interface for the proposed ACG-ingress and graph computational kernel with the storage system. Efficient representation of graph partitions is also a part of the HAL's design.

In addition to graph representation and partitioning, we further describe our plans for generating high-quality synthetic data sets for comprehensive performance benchmarking.

The subsequent part of this pipeline, the actual graph analytics computation, is a subject of the next design document. We, however, include our initial plans and progress on one of the use-cases that we have chosen to study.

2 Definitions

- **Arbitrarily Connected Graph (ACG)**: A graph with arbitrary edge relationships. The graph may be a tree, bipartite, undirected, directed, or any number of types. In any case, it will not be complete. Many graphs that model natural structures and real-world phenomena are arbitrarily structured. Many of them are scale-free, and some exhibit small-world and clustering characteristics.

- **ACG Ingress**: The process of constructing and loading an ACG into the exascale system. The ACG ingress process comprises the Big Data-ACG bridge, in this research. The graph will be constructed by applying extract and transform rules to large unstructured and semi-structured datasets.
- **Computational Kernel**: The application framework that supports the exascale structured machine learning and graph analytics. In this research, the computational kernel is based on GraphLab, an asynchronous distributed graph-parallel computational framework. GraphLab provides an in-memory data structure model, computational scheduling and synchronization, and a data consistency model.
- **Big Data Analytics (BDA)**: Big data analytics is the process of discovering latent patterns, understanding unknown correlations, or extracting meaningful information from data sets of which size and complexity are beyond the ability of traditional database management or data processing applications to process [1]. Some examples of big data include traffic sensory data (e.g., climate, and traffic), stock and commercial transactions, social interaction data, and digitalized media (e.g., pictures and videos).
- **Big Data Graph**: An ACG with associated "network information" derived from a Big Data corpus. The network information is largely comprised of arbitrarily-typed vertex and edge data.
- **Network Information**: Arbitrarily-typed data structures associated with vertices and edges.
- **Sub-Partition**: A graph-partition is typically represented as an ordered list (adjacency lists, edge lists etc.). A sub-partition is the further division of a partition into smaller portion. When a partition is too large for memory or thread processing, it may be divided into sub-partitions.
- **Synthetic Graphs**: Graphs that are artificially generated by human or computer.

3 Changes from Solution Architecture

Currently our project is not deviating from the path laid out in the solution architecture. However, in order to perform more comprehensive benchmarking we are proposing an additional step – generation of synthetic raw data.

The solution architecture laid out our plans for generating synthetic graphs and attaching network information to these synthetic graphs in order to stress test our system. However, that set-up will not be able to test ACG-ingress and the Big-data-HPC bridge. We realized that a more comprehensive way to test the ingress pipeline is to start with raw data sets, which is now part of our plan. Note that this inclusion is perfectly in line with our original intent.

4 Specification

This section details on various elements of the ACG ingress design. Section 0 introduces the HDF5 adaptation layer (HAL). Section 4.1 describes our graph partitioning strategy. Section 4.2 provides further details on the HAL and graph representation in HDF5. Section 0 outlines our plans on generating synthetic datasets and graphs. Section 4.6 describes our large-scale topic modeling experiment—our first proof-of-concept use case.The HDF5 Adaptation Layer (HAL)

Figure 1 describes the architecture of the proposed Big data – HPC bridge, and how the HAL is situated in the overall context. On the ingress side, the HAL transforms the output of the ACG-ingress to HDF5 data format. In the HPC world the HAL loads graph-partitions and associated network information to efficiently feed the graph computation kernel. Graph Builder, the starting point in our ACG ingress, is already designed to run on a Hadoop cluster, and hence we chose to follow the same set up in our HDF5-adapted ingress as well. The actual graph analytics will run on the target exascale machine. Both clusters will interact with the storage system as depicted in Figure 1.

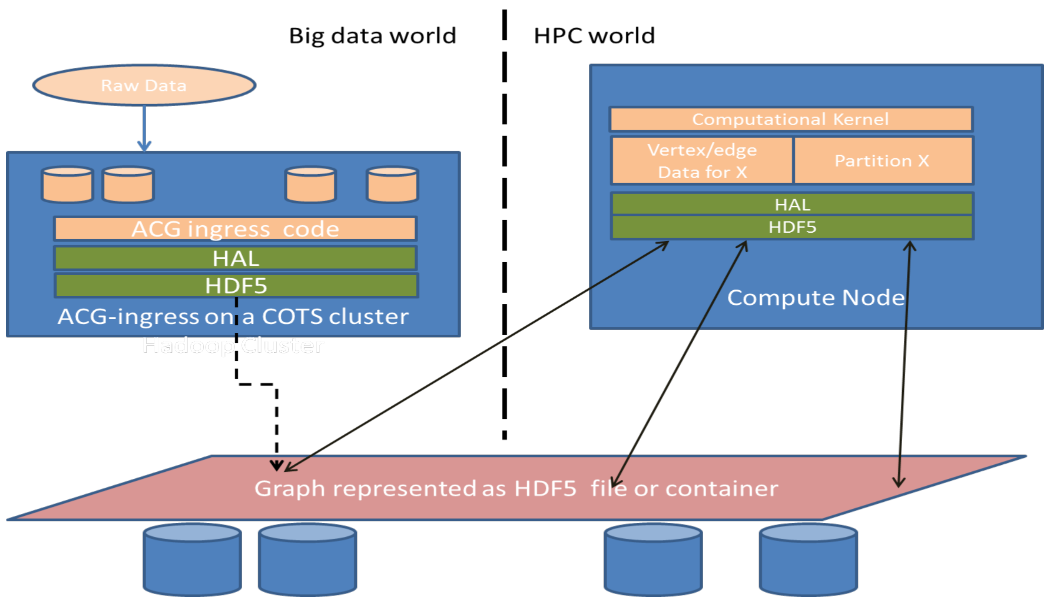


Figure 1 HAL helps establish the Big Data – HPC bridge by laying out data in HDF5 format

4.1 Graph Partitioning

4.1.1 Design Objective

Our design objective for an exascale graph partitioning algorithm is as follows. First, minimize the amount of communication between compute nodes by minimizing the edge-cut. Second, balance the number of edges in partitions to distribute the load across the compute nodes.

For the initial phase, we plan to use the partitioning algorithms outlined in Section 4.1.4 and Section 4.1.5. These schemes, natively available in Graph Builder, will help us bootstrap the bridge. We plan to experiment further with more state-of-the art algorithms, as described in Section 4.1.2 and Section 4.1.3, to further optimize the partitioning performance.

4.1.2 Choice of Algorithms

In general, graph partitioning is accomplished by finding patterns, such as cluster or community structure in the graph using spectral or topological analysis. Although spectral partitioning algorithms are known to produce very good partitions, the polynomial computational complexity ($O(n^{\omega}),\ 2 < \omega < 2.376$) of eigenvalue decomposition puts a scalability limit for their use on exascale graphs [2]. To resolve such scalability issues, a multi-level graph partitioning method has been introduced in [3].It works in three steps: (1) transform an input graph into a smaller graph, (2) apply the partitioning algorithm on the smaller graph, and (3) recover the original graph while maintaining the partitions. However, such a multi-level partitioning method relies on a global view of the whole graph structure and involves multiple steps running different algorithms. Recently, some researchers started looking into a new approach that does not require any global view of graphs. In this approach, a graph partitioning algorithm passes through an entire graph just once and partitions the graph on-the-fly [4, 5]. Especially, the one-pass graph partitioning algorithm described in [5] looks promising, in terms of edge cut, work load balance, and computational complexity; the authors claim that their proposed algorithm performs better than any heuristic one-pass graph partitioning algorithm and even achieves comparable performance to METIS [3].

4.1.3 Proposed Method: Cost Function based Graph Partitioning

Figure 2 shows the block diagram for the proposed one-pass graph partitioning. The proposed one-pass graph partitioning will scan the whole graph once and make online partitioning decisions on a per-edge basis. To make such an on-the-fly decision, we will design a cost for evaluating the cost of placing an edge to a certain partition.

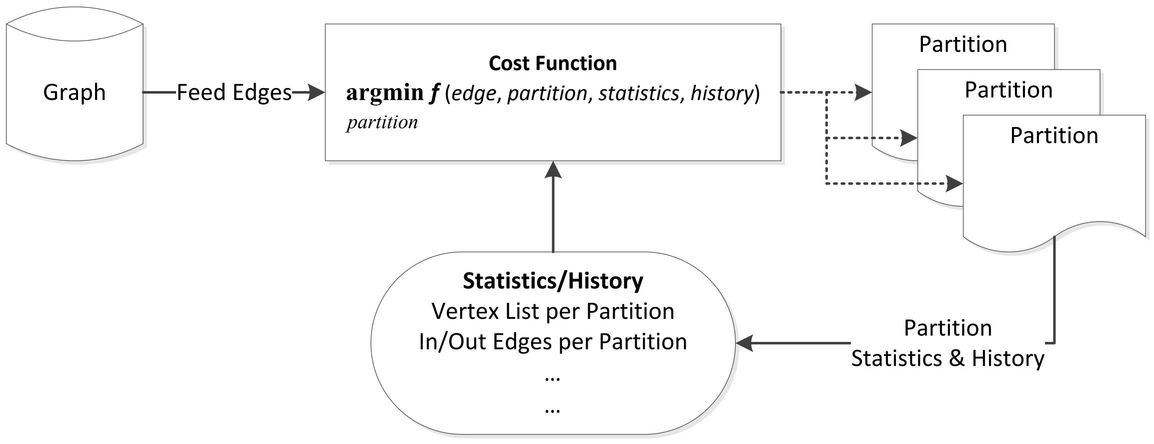


Figure 2. Block diagram for proposed one pass graph partitioning method.

Because the proposed method has no *a priori* knowledge on the whole structure of an input graph, it solely relies on statistics and history of previous partitioning decisions. For example, as it makes partitioning decisions on the edges and vertices, the method tracks topology-related statistical information, such as the number of in-edges and out-edges per partition, the number of vertices per partition, the size of each partition, *etc.*

4.1.4 Graph Builder Built-In Algorithm: Random Edge Assignment

Graph Builder also uses a one-pass built-in partitioning strategy, except it does not associate any cost with a sub-optimal placement. Each compute node of a Hadoop system places edges to a partition that is selected uniformly at random. Since the algorithm is not designed to minimize the edge cut between partitions, communication costs for graph computation may be sub-optimal.

4.1.5 Graph Builder Built-In Algorithm: Oblivious Greedy

Each compute node of a Hadoop system places edges using a greedy heuristic algorithm described in [6]. As the algorithm scans the edge list of a graph, it decides the partition an edge will be placed. The edge placement decision is based on the following four cases:

- Case 1: Both vertices of an edge have never been seen by the partitioning algorithm.
    - Randomly assign both vertices to a partition.
- Case 2: Both vertices have been seen by the partitioning algorithm and the two vertices are located on a single partition.
    - Assign to a partition that contains both ends.
- Case 3: Both vertices have been seen before but located on different partitions.
    - Assign to any partition that contains one of the two ends.
- Case 4: Only one vertex has been seen before.
    - Assign to a partition that contains one of the two ends.

For example, assume that a compute node is running the algorithm on the shard depicted in Figure 3 to divide the shard in two partitions. As the algorithm scans the shard of the edge list, it makes edge placement decision as depicted in Figure 4 through Figure 8.

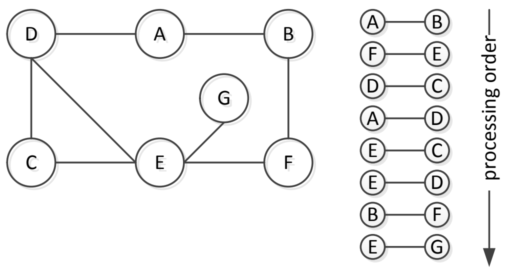


Figure 3. Compute node 1's shard.

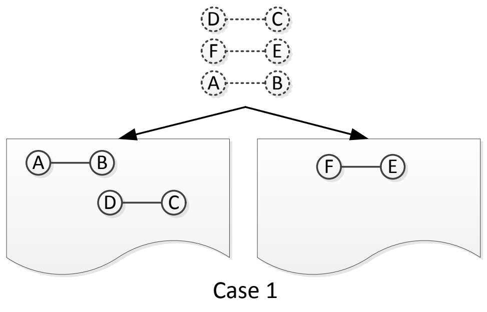


Figure 4. Partitioning decision: Case 1.

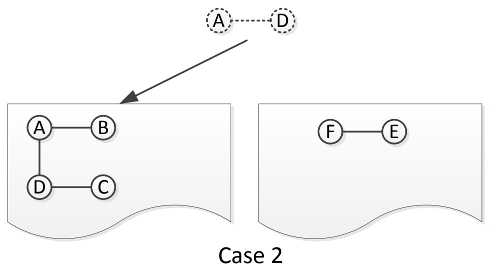


Figure 5. Partitioning decision: Case 2.

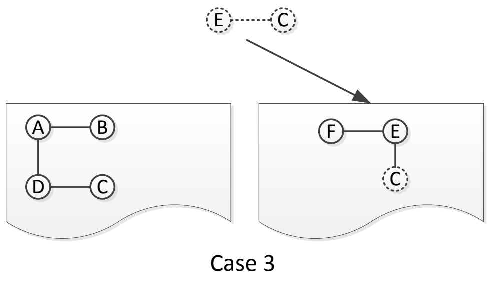


Figure 6. Partitioning decision: Case 3.

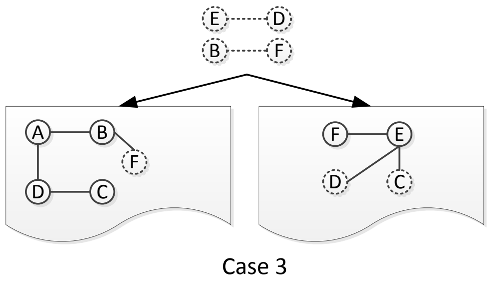


Figure 7. Partitioning decision: Case 3.

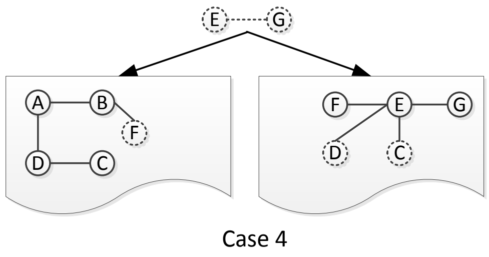


Figure 8. Partitioning decision: Case 4.

4.2 Graph Representation

In this document we only describe how the HAL lays out an ingress output graph in the HDF5 format. The consumption of this HDF5 object by the computational kernel will be described in the next design document to be delivered at a more advanced stage of the project.

To lay out the graph generated by the ACG-ingress, the HAL creates an HDF5 container. On a POSIX system the container maps to a file, while on the proposed exascale storage stack it will map to an IOD container. In addition to capturing the graph topology information, the HAL will also store partition information, so that a compute node can load the appropriate partition including its vertex and edge related data in memory.

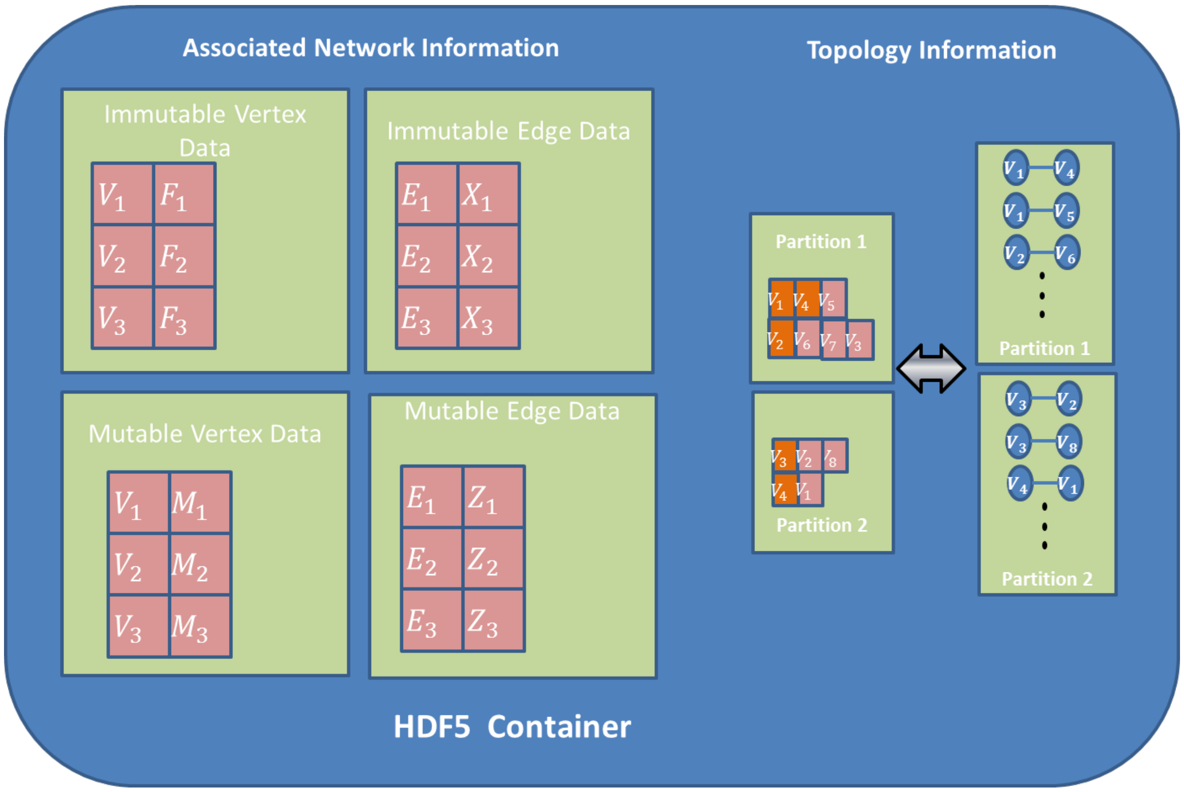


Figure 9 Proposed representation of a graph and its associated network information in a HDF5 container.

Figure 9 describes the initial design of an HDF5 container to represent a graph and its partitions. The whole graph will be represented in a single container (or file) so that information related to vertices, edges, and partitions can be accessed seamlessly from any compute node. Within the HDF5 container, the HAL creates the following types of representational units.

- **Immutable data objects**: These objects constitute the part of the network information that never changes during a graph-computation. However, in general they evolve over time. The immutable data objects can be associated with both vertices and edges. Examples of this type include name and gender attributes of individual profiles in a social network, the images and video files associated with them, and so on. This type of information can be small or substantially large. However, typically these are not heavily accessed during a graph computation. Inside an HDF5 container, these objects will be represented as HDF5 datasets.

- **Mutable data objects**: These are variables associated with vertices and edges that are actually part of a graph computation's update cycle. Inside an HDF5 container the mutable objects will be represented as attributes which are to be mapped to IOD KV objects underneath for efficient updates and retrieval.

- **Topology representation**: Topology is captured in separate structures inside the HDF5 container. The reason for this twofold. First, this lets the vertex programs inside the computational kernel access the graph topology without moving large network information. Second, it lets applications switch between different graph representations, again, without moving large amounts of data.

  Based on the algorithms, the HDF5 container will have different (possibly concurrent) representations of the topology. Figure 9 illustrates the interchangeable representations inside the container—both adjacency-list and edge-list representations are captured.

  Note that these two representations are fairly versatile and address the general needs of most of the algorithms in big data and HPC domains. They can directly represent both undirected and directed graphs. In the case of a directed graph, the in-edges and out-edges can be optionally kept in separate lists of identical structures.

- **Partitions and sub-partitions**: Partitions will be represented as either datasets or KV objects, or supported with both representations, subject to experimental findings.

  Once the partitions are created by the ingress, the vertices will be relabeled so that the ones belonging to a partition get a contiguous set of indices (See Figure 10). We are investigating multiple options for such a contiguous relabeling. For example, one way is to compute for a vertex its partition number and the offset within a dataset directly from its identifier by a quick arithmetic operation. Another option is to maintain, at the expense of more memory, both the partition number and the offset-index separately, to save the arithmetic operation. There are many other variations to this basic scheme.

  At any rate, the partition-friendly labels will ensure that the partition-id for any vertex can be derived completely in-memory (hence locally) without having to reach the storage or any other master node. The cost of re-labeling the vertex is incurred once, during the ingress. Translating the vertex labels back and forth, on the other hand, is an extremely quick in-memory operation.

  With the help of contiguously labeled vertices, HDF will lay out these partition structures as contiguously as possible on the underlying storage (POSIX File or IOD container), and therefore, retrieval will be faster, compared to scrambling together pieces from different parts of the storage.

  Sub-partitions are created on the fly, based on partition sizes, with respect to a compute node's memory. Their representation is similar to that of the partitions. They get sequentially loaded and unloaded (see Figure 11). As the lower levels of the I/O stack mature further, we plan on addressing efficient pre-staging of sub-partitions into the burst buffers on the IONs to boost the load-compute-unload cycle.

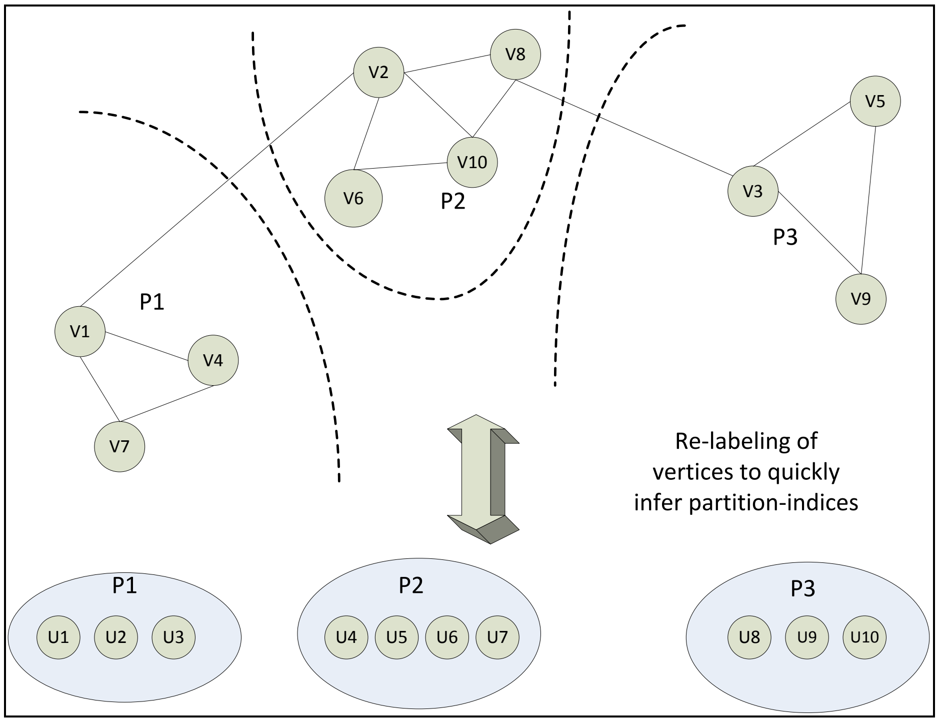


Figure 10. The vertices will be re-labeled such that vertices within each partition are labeled contiguously. The relabeling step will speed up distributed updates in the computational kernel as well as loading of partitions and sub-partitions.

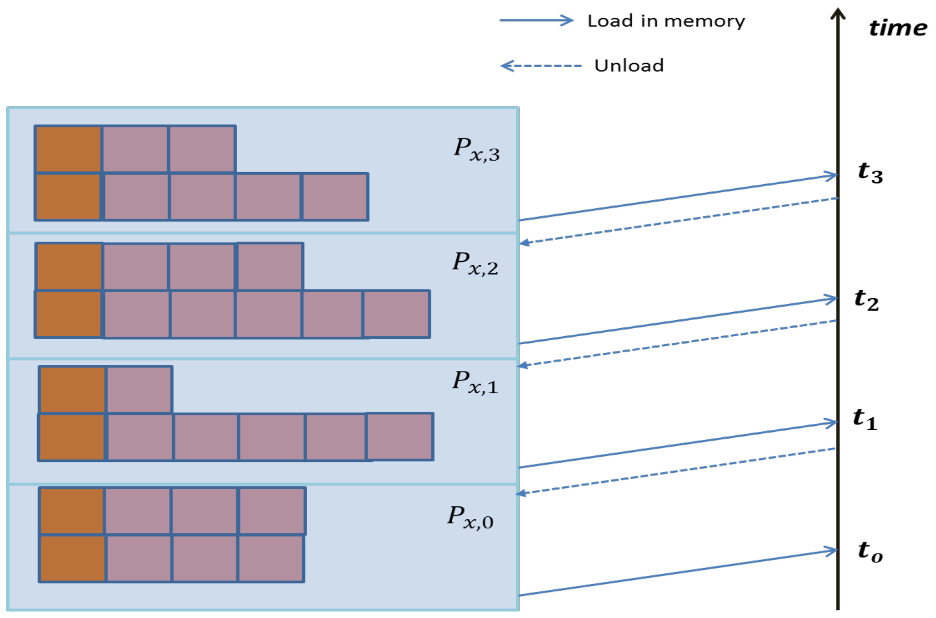


Figure 11 A partition P_x which is too big to fit in memory, is further sliced into sub-partitions ($P_{x,0}, P_{x,1}, P_{x,2}, P_{x,3}$) and loaded and processed in sliding-window style on a compute node.

4.2.1 Repartitioning of Graphs

One key design objective for graph representation is the ability to quickly repartition graphs. Repartitioning may be necessary for running successive rounds of analytics, possibly on a hardware configuration different from the original target.

By separating topology information from the rest of the network information, we can ensure that re-partitioning operations will be cleaner. As illustrated in Figure 9, another HDF group object can be created in the same HDF5 container of the graph; the new group will contain the topology information corresponding to another partition. And such partitions can be generated at any time.

4.3 Synthetic Datasets and Graphs

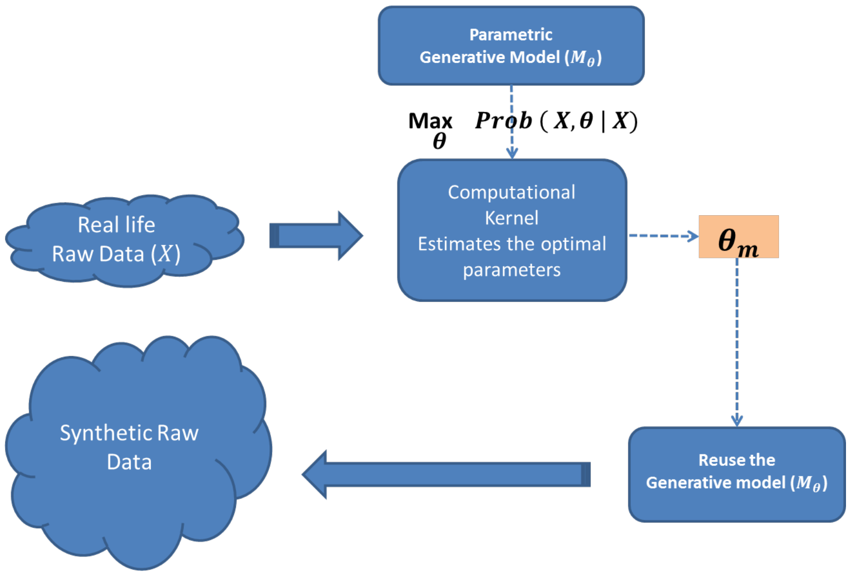
In order to corner-test our pipeline we need a very large collection of benchmarking data sets. However, it is not easy to obtain quality data sets of very large sizes. Hence we will synthesize our data sets in two different ways.

- Synthetic Raw data set: By generating synthetic raw data sets, we will be able to explicitly test the performance of the ACG-ingress. Generating raw data is comprehensive, but quite time-consuming, as it requires the ingress to execute every time before running graph computation.
- Synthetic Graphs: In order to directly execute the computational kernel (*i.e.,* bypass the ingress) we will generate synthetic graphs that mimic real-life characteristics.

4.3.1 Synthetic Raw Dataset Generation

Figure 12 describes our methodology for generating synthetic raw data sets. For a large class of graph analytics computation, the analysis algorithm assumes a probabilistic generative model (M_θ), where θ refers to a set of parameters that determines the model numerically. The algorithm eventually estimates θ by maximizing a likelihood function that best fits the problem. In Figure 12, this function is captured in the conditional probability distribution ($Prob\ (X, \theta \mid X)$), subject to maximization over the parameter space.

Once we run the graph-analytics algorithms on a graph generated from a real-life data set, and, in the process, recover the hidden parameter set ($\boldsymbol{\theta_m}$), we will run the generative model backward to generate synthetic data points that follow the distributions dictated by the parameter set θ_m. This allows us to generate arbitrarily-large data sets. We can slightly modify θ_m to generate variations of the original distributions, in order to create benchmarks for corner-testing.



Parametric Generative Model (M_θ)
Max θ $Prob\ (X, \theta \mid X)$
Computational Kernel Estimates the optimal parameters
Real life Raw Data (X)
θ_m
Synthetic Raw Data
Reuse the Generative model (M_θ)

Figure 12 Creation of a synthetic data set that mimics a real-life data set sample, but is much larger than the sample itself. A typical ACG computation in BDA setting often assumes a probabilistic generative model and estimates the underlying parameters. To stress-test the ingress performance, we will run the generative model backward using the parameters extracted from the real-life sample, and synthesize much larger data-sets.

4.3.2 Synthetic Power-Law Graph Generation

We use the stochastic Kronecker graph generation method in [7] to synthesize large-scale power-law graphs. Figure 13 shows the building blocks for generating synthetic large-scale graphs. The power-law graph model will generate a seed power-law graph, based on user inputs, including alpha-value and the size of the seed graph. Then, the stochastic Kronecker graph generator produces a large-scale synthetic graph with graph properties close to those of the seed graph.

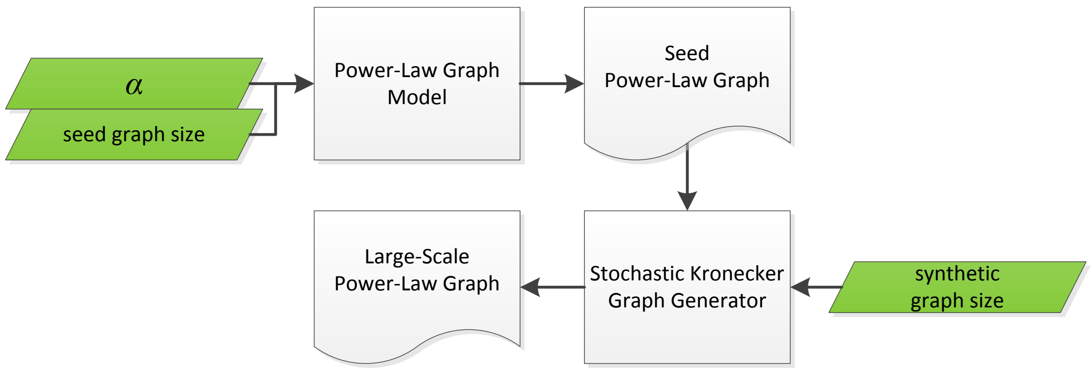


Figure 13. Building blocks for synthetic graph generation.

4.3.2.1 Power-Law Graph Model

Power-law degree distribution can be found in many natural graphs. For example, Figure 14 shows the degree distribution of the bipartite graph for Wikipedia Topic Modeling -with the power-law degree distribution parameter α = 2.23.

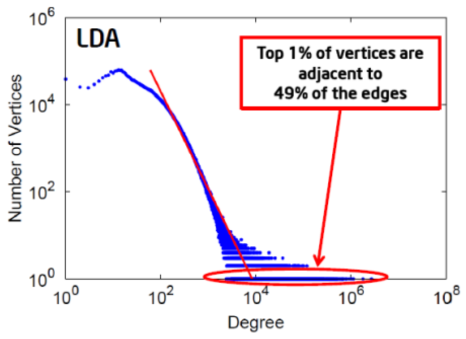


Figure 14. Power-law degree distribution of bipartite graph for Wikipedia topic modeling (α=2.23) [8].

To produce small power-law seed graphs, with tunable alpha values, we use the mathematical model to produce a power-law degree distribution. In the model, the probability of a vertex to have degree k is defined as $P(k) \sim k^{-\alpha}$. From the degree distribution, we will generate a random power-law graph which will be used as a seed graph for the Stochastic Kronecker graph generator.

4.3.2.2 Stochastic Kronecker Graph Generator on Hadoop

The Stochastic Kronecker graph generator is known to produce good power-law graphs and has been adopted by Graph500 benchmark to produce large-scale synthetic graphs. The stochastic Kronecker graph generator uses a likelihood optimization method described in [7] to produce the following 2-by-2 initiator matrix from the seed graph as follows:

$$\mathbf{K_1} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \ \ 0 \le a, b, c, d \le 1$$

Next, the stochastic Kronecker graph generator performs Kronecker multiplication on the initiator matrix until it reaches the desired size. Thus the *n*-th Kronecker product of initiator matrix forms the recursive multiplication,

$$\mathbf{K}_2 = \mathbf{K}_1 \otimes \mathbf{K}_1 = \begin{bmatrix} a\,\mathbf{K}_1 & b\,\mathbf{K}_1 \\ c\,\mathbf{K}_1 & d\,\mathbf{K}_1 \end{bmatrix},$$

$$\mathbf{K}_3 = \mathbf{K}_1 \otimes \mathbf{K}_2 = \begin{bmatrix} a\,\mathbf{K}_2 & b\,\mathbf{K}_2 \\ c\,\mathbf{K}_2 & d\,\mathbf{K}_2 \end{bmatrix},$$

$$\ldots$$

$$\mathbf{K}_n = \mathbf{K}_1 \otimes \mathbf{K}_{n-1} = \begin{bmatrix} a\,\mathbf{K}_{n-1} & b\,\mathbf{K}_{n-1} \\ c\,\mathbf{K}_{n-1} & d\,\mathbf{K}_{n-1} \end{bmatrix}.$$

The *n*-th Kronecker product is the probability matrix where each matrix element is the edge probability between a pair of vertices signified by row and column indices. We will use the Hadoop cluster to parallelize the stochastic Kronecker graph generation process. Since obtaining the *n*-th Kronecker product is embarrassingly parallel [9], parallelized graph generation using the Kronecker product is easily parallelized as well.

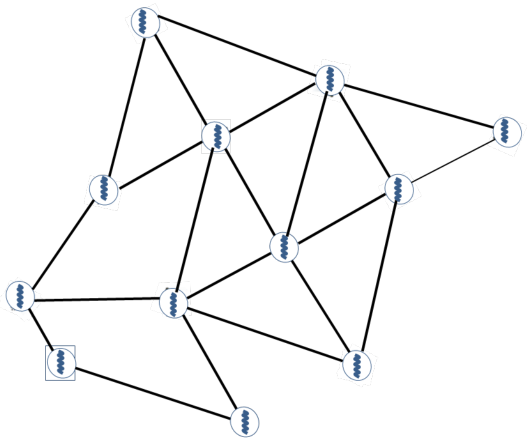


Figure 15. Graph-parallel computation model. There is a computing process associated with every node in the graph, and the processes asynchronously communicate with the neighboring vertices.

4.4 Computational Kernel for Graph-Parallel Computing

4.4.1 Overview of vertex programs

Many algorithms in the context of large-scale data mining and machine learning can be modeled as graph-parallel computation. The key component in this paradigm is a *vertex program* (VP). A VP is a single unit in a parallel program that can be thought of running over a vertex. Each VP communicates with other VPs that run on neighboring vertices. Typically the structure of an ACG is reflected in its computational graph (see Figure 15. Graph-parallel computation model. There is a computing process associated with every node in the graph, and the processes asynchronously communicate with the neighboring vertices.

There have been many different models of graph-parallel computers that could be chosen as our computational kernel. We chose GraphLab [6] because of its support for asynchronous messaging, and data-consistency support.

4.4.2 Data consistency model

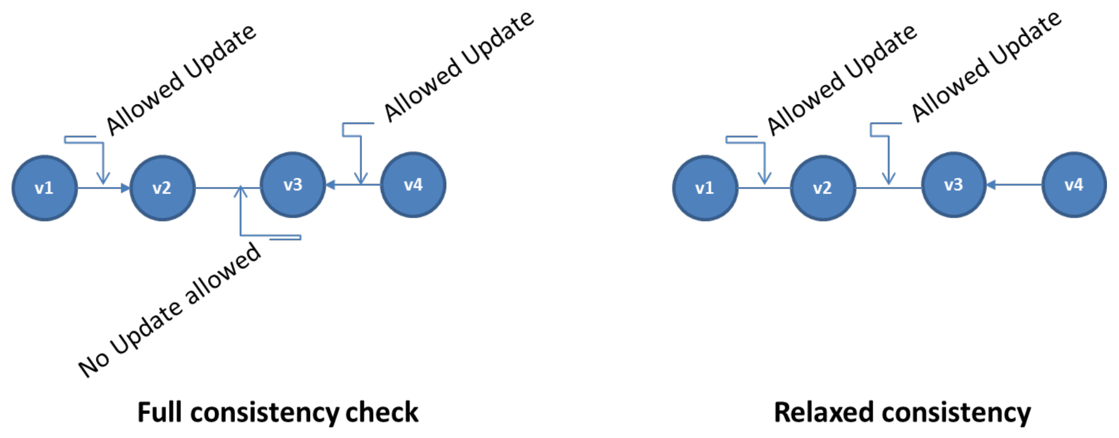


Figure 16. Consistency checking in the computational kernel. In Full consistency check, all parallel updates are separated by at least two vertices. This condition is strict and eliminates all race conditions (left). In a more relaxed consistency model (known as Edge-consistency), two updates can happen in parallel if they are not colliding on a single edge (right). The edge-consistent updates run faster, but are prone to more race conditions.

One important aspect of any parallel program is maintaining data consistency. The nature of the algorithms in the space of BDA and LSML are statistical in many cases; this means that strict data consistency may not be always mandatory. However, it has been established that ensuring data consistency may improve the convergence rate of algorithms. At the same time, ensuring consistency is more restrictive—since many parallel programming units need to be blocked, in order for others to proceed. Fortunately, GraphLab offers multiple levels of consistency checks to be imposed by the applications.

Figure 16 shows two possible levels of consistency checking. In a full consistency model (left), no two vertex programs that can potentially conflict on their updates can run at the same time. The only choice therefore is to allow VPs to proceed simultaneously only if they are separated by a path of at least length two. In a somewhat relaxed consistency model (right), two vertices are allowed to proceed if their updates do not collide on an edge. That means two VPs separated by at least one vertex can now proceed in parallel. There are other relaxed consistency models that can be implemented, however, since this is not the principal theme of our work, we do not discuss them in detail this document. We note that our key design objective is to support the data consistency models while we port this library over our exascale stack.

4.5 GraphLab modification for HDF5

In order for GraphLab to work on our data representation, we have to intercept the computational kernel at multiple points.

1. **Data Translation and access**: Data translation from the HDF5 format to the native in-memory representation used by GraphLab (and vice versa) is our primary (and mandatory) requirement. We'll intercept the data load/unload module(s) of GraphLab to do this bidirectional representational switch. We will experiment with different mechanisms of retrieval of network information that are required by the algorithms to make their execution more efficient.

We also wish to extend support for the following aspects, based on the readiness of the exascale I/O stack:

2. **Out-of-core support**: This pertains to the situation when the size of the computational graph in question is too big fit in-memory; even after partitioning a graph, each partition needs to be split into sub-partitions that need to be paged in and out as needed. In such situations, the updates cannot be all be handled in-memory anymore, and the computational kernel will be intercepted by HAL to translate the change all the way to the storage.
3. **Consistency check**: This only pertains to the vertices (and edges) that are loaded in-memory, and therefore can be maintained as such even for out-of-core computing. However, in order to compute which VPs are allowed to run, only in-memory vertices need to be counted, and all the vertices that remain out-of-core should be excluded.
4. **Transactional bundling**: This especially applies to out-of-core graph computing, as well as for fault tolerance. Many of the updates that need to reach storage will be bundled as all-or-none into a single transaction. The examples of these are the collections of updates performed over an entire sub-partition.
5. **Data-usage hints**: By collecting statistics on which partitions are more likely to be brought into memory, applications will generate hints for the layer below to keep special blocks, such as special sub-partitions containing highly-connected vertices always in-memory, so that other vertices can quickly perform their updates and retrieval.

Figure 17 portrays a high-level view of out-of-core graph computing, in the event that all desirable requirements are implemented over the exascale stack. At any point in time, there will be a set of graph-partitions loaded in-memory; however, asynchronous updates will be made and these updates will be consistently first sent to the burst buffer, and then migrated to the storage to be viewed by other VPs at different point of execution. Similarly, different partitions of a graph will be pre-staged onto the NVRAM to reduce computing latency.

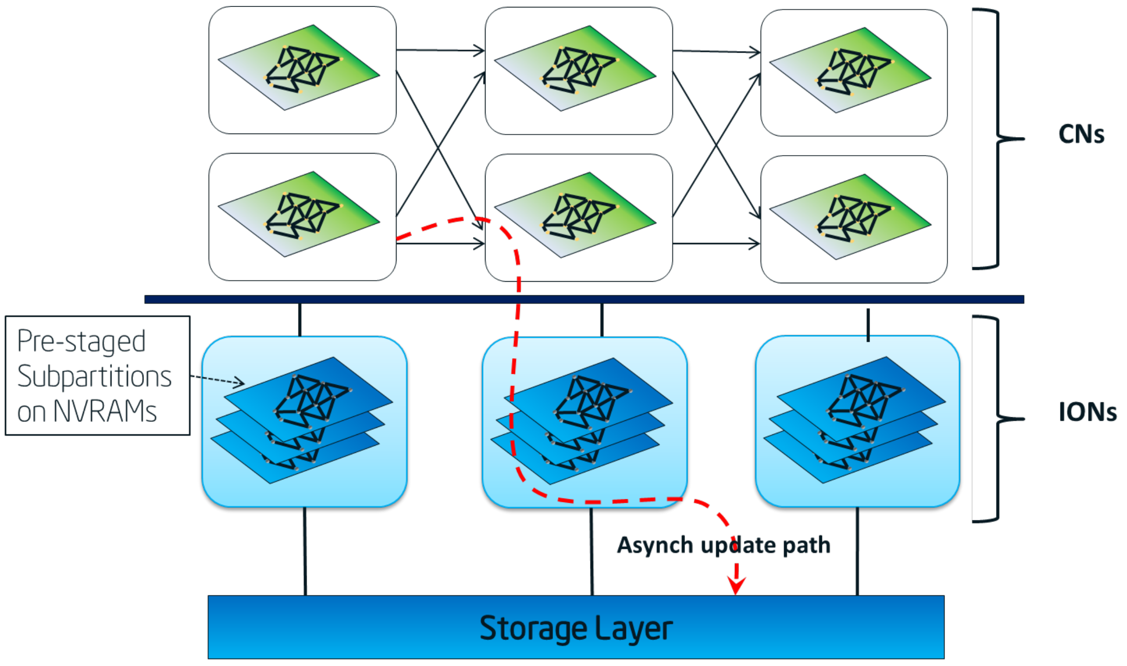


Figure 17. Complete picture of graph computation over the exascale stack. Different partitions are paged in and out from the storage to the compute nodes.

4.6 Use Case: Topic Modeling

To test every step in our computational pipeline, we will run a large-scale probabilistic graphical modeling experiment, in which we conduct topic modeling on one or more large corpora of textual data. These experiments will provide two primary benefits. First, they will demonstrate that our system is able to carry out real-world computation over a large-scale ACG. Second, they will provide us with initial benchmarks for the performance of our system on this particular problem, allowing us to further improve our topic modeling algorithms for scaling up to exascale computations. Using the right data set (i.e., one containing gold-standard, expert-curated topic assignments), a third benefit would also be the ability to assess the accuracy of our topic modeling algorithms.

4.6.1 Dataset Selection & Acquisition

We have several considerations in selecting our real-world data set. Of primary importance to us is size: the dataset we use here must be big enough to test our graph partitioning algorithms and storage architecture. Another consideration was accessibility: the data must be freely-available to the public, either under an open source licensure (e.g., Creative Commons), or released as publically-accessible in some other manner (e.g., fair use of public records). Finally, since we are pushing the boundaries of what has been done with topic modeling algorithms, it would be desirable if our dataset had some inherent topic-like structure to it that we could leverage to evaluate the accuracy of our algorithms. Along these lines, although a synthetic dataset would give us the flexibility to arbitrarily scale up the data size, we prefer to use a real-world dataset, as it will be easier to get an intuitive sense of the accuracy of our topic modeling algorithm, even if we have to compromise somewhat on the size. Taking all this into account, we narrowed the datasets we considered down to four: ClueWeb09-English, Google tri-grams, MEDLINE, and English Wikipedia (Table 1).

Table 1. Descriptive statistics for the document corpora considered for use in the exascale topic modeling experiment. The dataset we selected for the presently-described experiment is highlighted in green.

Data Set	Open Source	Uncompressed Size	Number of Documents
ClueWeb09-English	Yes	13.4 TB	$5.03\ x\ 10^{8}$
Google tri-gram	Yes	218.1 GB	$2.45\ x\ 10^{10}$
MEDLINE	Partially	72 GB	$1.35\ x\ 10^{7}$
English Wikipedia	Yes	38 GB	$1.2\ x\ 10^{7}$

Each of these datasets has certain positives and negatives associated with its use. We will discuss these in turn, to describe how we arrived at selecting the ClueWeb09 dataset. One positive aspect of the Google tri-gram dataset is that it is frequently used to evaluate text-mining and information retrieval systems. Although carrying out a topic modeling experiment on this set would be somewhat meaningless, if we were to use it for another type of textual analytics study, we could potentially have other published research against which we could compare our results. The MEDLINE dataset, which consists of every MEDLINE record on the National Library of Medicine's PubMed search engine (http://www.ncbi.nlm.nih.gov/pubmed) is relatively small, but it contains semi-structured documents that have been manually labeled with MeSH terms (Medical Subject Heading terms) by expert curators, which would allow us to easily assess the validity of our topic modeling results. The dataset, though freely-available through the NLM's search engine, is not easily obtained, however. Finally, the ClueWeb09-English corpus consists of the English language subset of ClueWeb09, a data collection consisting of the html of websites obtained from one year of web crawling by a group at Carnegie Mellon University. 13.4 TB of textual data will be a nearly large enough dataset for our testbed, and it will allow us to get a sense of the efficiency of our pipeline. One downside of the ClueWeb09-English is that there are no pre-existing topic labels associated with the documents it contains. To address this, we will use other methods of evaluating topic modeling algorithms, such as perplexity, and manual examination of the most-common words used in documents contained in the various topics identified by our approach. In addition, other researchers have already analyzed subsets of the ClueWeb09-English corpus, which we may be able to use to compare with our results. Ideally, we would have liked to use a much larger dataset (e.g., one the petabyte-level), to get a sense of how data of this magnitude is handled by our system. We will continue to search for a larger real-world dataset for which a topic modeling experiment would be meaningful; in the meantime, we will generate synthetic datasets and graphs for stress testing our framework. In the meantime, our plan is to first test our pipeline using the English Wikipedia data set. This collection has already been extensively studied, which will allow us to replicate results that have been observed by other researchers at Intel. Next, we will perform the same topic modeling studies using the MEDLINE dataset, which will allow us to evaluate the accuracy of our algorithms. Finally, we will conduct a topic modeling experiment on the ClueWeb09-English collection, which will allow us to assess the scalability of our pipeline.

4.6.2 Topic Modeling Algorithm

There are several approaches to topic modeling that are available to us, including *k*-means clustering, Latent Semantic Indexing (LSI), and Latent Dirichlet Allocation (LDA). For testing our pipeline, we opted to use LDA. There were several reasons for this, the most prevalent of which was that LDA has already been implemented for graph data in the open source GraphLab software.

5 Open Issues

- Very large scale data sets occur very commonly, however, because of their proprietary nature it is not easy to obtain them at will. We will constantly seek bigger real data sets and will make use of them as we deem fit.
- We are also investigating the possibility of testing the exascale stack with other graph-based application(s). In particular, following the earlier reviews, we have engaged with a group of LBNL researchers and are in the process of evaluating an astronomy related application involving large-scale statistical inference over graphs. However, using the application in the scope of the current project is contingent on various factors, such as its readiness with respect to the timeframe of this project, and its suitability to run as a graph-parallel program over our computational kernel.
- As we gain more insight into various aspects of HDF5 ACG-representation, such as programmability, memory usage, storage performance, and so on, the exact design is expected to evolve.
- Synthetic Data set generation – our plan will work only for situations for which there is a probabilistic parametric generative model.
- The proposed HDF5 representation does not capture hypergraphs directly. Hypergraphs are generalizations of graphs, and are used in a limited, albeit important, set of data analytics applications. Whether we investigate appropriate representation for these structures will be decided based on the exact needs of the use-cases to be finally implemented.

Reference

[1] "Big Data," 11 1 2013. [Online]. Available: http://en.wikipedia.org/wiki/Big_data.

[2] J. Demmel, I. Dumitriu and O. Holtz, "Fast linear algebra is stable," *Numerische Mathematik,* pp. 55-91, 2007.

[3] G. Karypis and V. Kumar, "Metis-unstructured graph partitioning and sparse matrix ordering system, version 2.0," 1995.

[4] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2012.

[5] C. E. Tsourakakis, C. Gkantsidis, B. Radunovic and M. Vojnovic, "FENNEL: Streaming Graph Partitioning for Massive Scale Graphs," Microsoft Research, 2012.

[6] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. of the 10th USENIX conference on Operating systems design and implementation, OSDI*, 2012.

[7] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *The Journal of Machine Learning Research,* vol. 11, pp. 985-1042, 2010.

[8] T. L. Willke, "Large-Scale Machine Learning Challenges," Intel, 2012. [Online]. Available: https://01.org/graphbuilder/documentation/large-scale-ml-challenges. [Accessed 25 2 2013].

[9] "The Graph 500," [Online]. Available: www.graph500.org. [Accessed 25 2 2013].

[10] "Machine Learning," Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Machine_learning. [Accessed 10th January 2013].