Billion node graph inference: iterative processing on The Machine

Chen,Fei; Gonzalez, Maria Teresa; Viswanathan, Krishnamurthy; Cai, Qiong; Laffite, Hernan; Rivera, Janneth; Mitchell, April; Singhal, Sharad

Hewlett Packard Labs

HPE-2016-101

Keyword(s):
The Machine; probabilistic graphical models; probabilistic Inference; graph processing; non-volatile memory; malware detection

Abstract:
Iterative graph processing has emerged as a necessary and powerful tool at the heart of solutions to several large scale analytics problems such as malware detection, genome analysis and online advertising. Due to the large amount of communication involved, iterative graph processing on large graphs does not scale well on distributed systems. To demonstrate the benefits of a memory driven computing architecture, we designed and implemented an iterative graph processing engine that uses The Machine's Fabric-attached-memory as a communication medium. The engine demonstrates near-linear scalability with a 162x speed-up over a state of the art graph processing system on a Superdome X and a projected 85x speed up over the same system on the Machine Fabric Testbed (MFT).

External Posting Date: December 22, 2016 [Fulltext]
Internal Posting Date: December 22, 2016 [Fulltext]

Billion node graph inference: iterative processing on The Machine

Fei Chen, Maria Teresa Gonzalez, Krishnamurthy Viswanathan, Qiong Cai, Hernan Laffite, Janneth Rivera, April Mitchell, Sharad Singhal

Hewlett-Packard Labs

Abstract

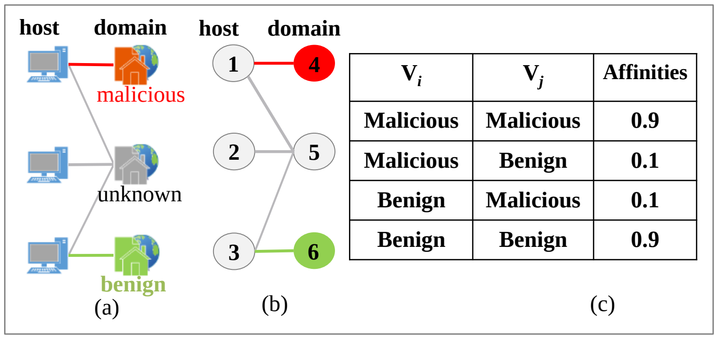
Iterative graph processing has emerged as a necessary and powerful tool at the heart of solutions to several large scale analytics problems such as malware detection, genome analysis and online advertising. Due to the large amount of communication involved, iterative graph processing on large graphs does not scale well on distributed systems. To demonstrate the benefits of a memory driven computing architecture, we designed and implemented an iterative graph processing engine that uses The Machine's Fabric-attached-memory as a communication medium. The engine demonstrates near-linear scalability with a 162x speed-up over a state of the art graph processing system on a Superdome X and a projected 85x speed up over the same system on the Machine Fabric Testbed (MFT).

I. Introduction

Representing data as graphs has proven to be effective for solving a class of analytics problems. In some cases, such as social networks the graph is apparent. In others cases such as DNA analysis it is derived. Traditionally, graph structure have been used to answer questions about connectivity and path length. However, more powerful insights can be obtained by combining tools from probability with the structure of the graph. This combination, called a graphical model, can be used to infer hidden characteristics of variables represented in the graph.

For example, consider the problem of detecting in real time malicious domains accessed by an enterprise's hosts and exhibited through billions of HTTP proxy logs given a small list of known malicious and benign domains (Figure 1 (a)).

The graphical model to this problem is illustrated in Figure 1(b). Each vertex represents a binary-valued variable associated with either a server or domain that indicates if the particular server or domain is malicious. An edge between a server and a domain exists if the corresponding server has made a request to the corresponding domain. Besides the structure, the graphical model also encodes the extent of the dependency between the variables through metadata associated with vertices and edges. An example of such metadata is an edge factor table (shown in Figure 1(c)) that quantifies the "dependencies" between the variables connected by the edge. Through this graphical model one can infer the probability that a specific domain is malicious given a small list of malicious and benign domains through a technique called Gibbs Sampling (GS). In this method, a binary state variable is associated with each vertex in the graph and the state of each vertex is updated iteratively based on the state of its neighbors and the edge factor tables until the fraction of times the state variable takes a certain value converges.



V_i	V_j	Affinities
Malicious	Malicious	0.9
Malicious	Benign	0.1
Benign	Malicious	0.1
Benign	Benign	0.9

Figure 1 Detecting malicious domains.

Almost all solutions to problems of this nature have the following characteristics: a) aspects of the graph such as vertices or edges have state variables associated with them, b) these state variables are updated in an iterative fashion, c) the state updates of a variable depend on the variables in a neighborhood as described by the graph, and d) inferences are computed using functions that take the state variables as input. Some famous algorithms that fit into this paradigm include PageRank (used to rank web pages), Label Propagation (used for social network analysis), the above mentioned Gibbs Sampling (GS) and Belief Propagation (BP) (used for problems such as malware detection and click-through-rate prediction in online advertising). However, these algorithms are computationally expensive especially when the graph sizes are large. For example, Facebook social network size is over a billion of people and the friendship links go easily to a trillion of relations, the malware detection graph is over 100's Millions on computers and their links go to billions of connections, online advertising involved 100s of millions of people and millions of transactions a day. At the same, these applications demand fast time response. For example, in the security domain, the response time is often "real-time" to detect malware within minutes and stop the spreading. The current solutions on large scale graph inference cannot meet such application requirements (see the experiment results).

Given the power and the wide applicability of these techniques, a computational engine for iterative computation on graphs is well motivated. However, one of the main challenges of the graph processing is scalability. The number of computations involved in each iteration of such processing is proportional to the number of edges in the graph. In order to continue to produce results in a timely fashion, a large number of CPU cores are required as the graph gets larger. Implementing this on a distributed system steadily increases the amount of communication over LAN with associated synchronization overheads, and severely limits the scaling as more machines are added. The Machine's Fabric-attached-memory (FAM) on the other hand, provides a fast medium to asynchronously communicate the states of the variables with reduced latency. Thus, we constructed an iterative graph processing engine that can take advantage of the FAM for fast communication. We also evaluated its performance on today's hardware as well as projected its performance on The Machine.

II. Background

We first provide a brief introduction to graphical models. Then, we describe the probabilistic inference algorithms to highlight the nature of data processing involved and encapsulated as part of our solution.

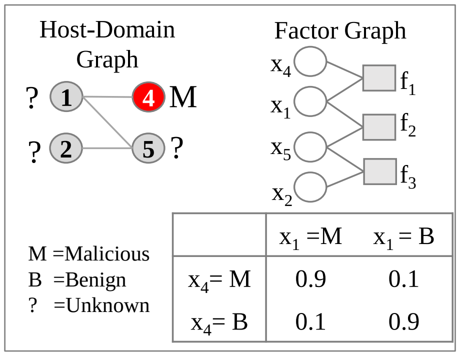
Graphical models are popular tools for representing probability distributions on random variables. Given a collection of random variables, in the most general case, they may not be independent of each other. In other words, their joint distribution cannot be factored into a product of their marginals. This makes it hard to represent and interpret the relationships between the random variables. For example, if one expressed the joint probability distribution of *k* random variables over a set D as a table of *k*-tuples and their associated probabilities, then such a table would take $O(|D|^k)$ space which is exponential in the number of random variables. Fortunately, in many real world applications, the dependencies among random variables is more structured or can be modeled as such. Graphical models are probabilistic models where graphs are used to represent the conditional independence between random variables. There are many types of graphical models, including Bayesian Networks, Markov Networks and Factor Graphs. We focused our work on **Factor Graphs,** although it can easily be extended to other models.

A **factor graph** is a bipartite graph which specifies how variables are dependent on each other. Specifically, a factor graph is an undirected graph containing two types of vertices: variables vertices and factor vertices. Each variable vertex corresponds to a random variable *x*, and each factor vertex corresponds to a *factor f*. Edges exist only between variables vertices and factor vertices. A factor *f* is connected to a set of variables $x_1, x_2, \ldots, x_k$, indicating these variables are dependent. The strength of this dependence is parameterized by a *potential function* $\phi_f: D^k \rightarrow R^+$ that is associated with the factor *f*, where *k* is number of variables involved in the factor. We refer to the set of variables $x_1, x_2, \ldots, x_k$, connected to *f*, the *scope* of *f*, and denote it by *S*(*f*).

Given a factor graph on a set of variables $x_1, x_2, \ldots, x_n$, a set of factors $f_1, f_2, \ldots, f_m$, and its associated potential functions $\phi_1, \phi_2, \ldots, \phi_m$, the joint distribution on all variables $x_1, x_2, \ldots, x_n$ is proportional to the product of all potential functions . Formally,

$$p(x1, x2, \ldots, xn) = \frac{1}{Z} \prod_{i=1}^{n} \phi_i(S(f_i)),$$

where Z is a normalization constant. In other words, factor graphs express the factorization of the joint distribution of random variables into factors involving subsets of the variables.



	x_1 =M	x_1 = B
x_4= M	0.9	0.1
x_4= B	0.1	0.9

Figure 2. A factor graph of detecting malicious domains

Figure 2 illustrates a factor graph resulting from the problem of detecting malicious domain. Each host and domain are associated with a random variable *xi* indicating if the host/domain is malicious or not. We can further model the dependency among these variables as a pairwise dependency between labels of adjacent host/domains. This results in four variable vertices $x_1, x_2, x_3. x_4$, which are connected to three factor vertices f_1, f_2 and f_3. Let ϕ_1, ϕ_2 and ϕ_3 be the corresponding potential functions. Then, from the graph it follows that the joint distribution of the 4 variables can be written as

$$P(x_1, x_2, x_4, x_5) = \frac{1}{Z} \phi_1(x_1, x_4) \cdot \phi_2(x_1, x_5) \cdot \phi_3(x_5, x_2)$$

where *Z* is the normalization constant.

Factor graphs encode the conditional dependencies implied by the factorization of the joint distributions in an elegant manner. We elaborate below. The *Markov Blanket* of a variable x_i, denoted as $mb(x_i)$ is defined as all variables which share at least one factor with *xi*. Formally,

$$mb(x_i) = \{x_j \mid j \neq i, \exists f \in F \, such \, that \, x_i \in S(f) \wedge x_j \in S(f)\},$$

where F is set of all factors in the graph. Conditioned on its Markov Blanket $mb(x_i)$, x_i is conditionally independent of all other variables in the graph. For the example in Figure 2, the Markov Blanket of x_4 is $mb(x_4) = \{x_1\}$ since x_1 is the only other variable that shares a factor with x_4. Therefore, conditioned on x_1, x_4 is independent of x_2 and x_5.

Probabilistic Inference Algorithms

There are two basic types of inference tasks in graphical models. The first task is to infer the marginal distributions of individual variables, i.e. to compute $p(x_i)$. The second task is to infer the maximum likelihood assignment of all variables, i.e. to compute $x_1^*, x_2^*, \ldots, x_n^* = argmax_{\{x_1, x_2, \ldots, x_n\}} p(x_1, x_2, \ldots, x_n)$, the assignment that maximizes the joint probability. In this work, we consider both types of inference tasks. Solving the marginalization task exactly is often intractable for large graphical models. Therefore, in large graphical models which involve many variables, approximate inference algorithms are employed. The inference algorithms employed can be classified into two broad classes - sampling based algorithms and message passing based algorithms.

Sampling Based Inference and Gibbs Sampling. Broadly speaking, sampling based algorithms perform inference by sampling at each vertex from D, the set of values taken by the variable, based on the potential functions. We describe one such algorithm, Gibbs Sampling (GS), in greater detail. GS generates a sequence of samples according to the joint distribution encoded by the graphical model. The key observation underlying GS is that while sampling directly from the joint distribution is hard, sampling based on the conditional distributions is easy. Particularly since conditioned on $mb(x_i)$, x_i and all other variables are independent, i.e.,

$$p\left(x_i \middle| x_{1,} \ldots, x_{\{i-1\}} x_{\{i+1\}} \ldots x_n\right) = p(x_i | mb(x_i)).$$

GS starts by initializing all variables with a random value in D. Then it sequentially sweeps over all variables to generate a new sample. For each variable *x*, it first reads the current values of all the variables in $mb(x)$. It then computes the conditional distribution

$$p(x|mb(x)) \propto \prod_{fi \in N(x)} \phi i,$$

where *N*(*x*) is the set of neighboring factors of *x*. Finally, it generates a new sample value for *x* according to this conditional distribution. Upon convergence, the marginal distribution of *x* is estimated based on the samples generated for *x* (the empirical distribution of the samples). The most likely assignment is estimated as the element from D that occurs most often in the sample.

Message Passing Inference and Belief Propagation. In Belief Propagation (BP), each variable *x* maintains a "belief", which is its current estimate of its marginal distribution. Based on its current belief, it then sends "messages" to its neighbors in *mb*(*x*), which capture the component of the belief that is not based on that particular neighbor's past message input. Specifically, BP starts by initializing belief and messages on all variables and edges. Then at every iteration, each vertex sends a message to each of its neighbors, receives a message from each of its neighbors and updates its belief. The messages are mappings from D to real numbers. So one can view the messages as tuples of real numbers. A message from a vertex associated with variable *x* to its neighboring vertex associated with factor *f*, is denoted by $m_{x \to f}$ and is the product of the messages from all other neighboring factor vertices, except the one associated with *f*. Formally,

$$\forall v \in \mathrm{D}, m_{x \to f}(v) = \prod_{f' \in N(x)\{f\}} m_{f' \to x}(v) \qquad (1)$$

A message from a vertex associated with factor *f* to its neighboring vertex associated with variable *x* is the product of the factor and the messages from all other neighboring variable vertices, marginalized over all variables except the one associated with *x*. Formally,

$$\forall v \in \mathrm{D}, m_{x \to f}(v) = \sum_{S(f)\{x\}} \phi(S(f)) \prod_{x' \in S(f)\{x\}} m_{x' \to f}(v) \qquad (2)$$

Finally, the belief of a variable *x* is the product of all incoming messages. Formally,

$$\forall v \in \mathrm{D}, b_x(v) = \prod_{f \in N(x)} m_{f \to x}(v) \qquad (3)$$

Upon convergence, the belief of each variable converges to a value proportional to the marginal distribution of that variable. This iterative algorithm is called *sum-product* algorithm. To obtain, instead, the most likely assignment inference, we can substitute the sum operator with the max operator in (2). This is called the *max-product* rule. As f with variables *x*, for each factor *f* we can define an analogous "belief" quantity *bf* where for all $v \in \mathrm{D}$,

$$b_f(v) = \prod_{x' \in S(x)} m_{x' \to f}(v) \qquad (4)$$

Then, we can express all the messages in terms of beliefs as follows. Equation (2) can be written as

$$\forall v \in \mathrm{D}, m_{x \to f}(v) = \frac{b_x(v)}{m_{f \to x}(v)} \qquad (5)$$

and Equation (5) can be written as

$$\forall v \in \mathrm{D}, m_{f \to x}(v) = \sum_{S(f)\{x\}} \phi(S(f)) \frac{b_f(v)}{m_{x \to f}(v)} \qquad (6)$$

It is important to notice that from the computational point of view, GS mainly accesses information associated with vertices, whereas BP accesses information associated with both vertices and edges.

III. Our solution

We have designed and tested an iterative graph processing engine that exploits the large memory bandwidth provided by The Machine's FAM. Before we detail our contributions, we describe what the engine can do. The engine takes a graph and associated metadata as input and performs the following computation. Each vertex is associated with a state variable. The state variable is updated

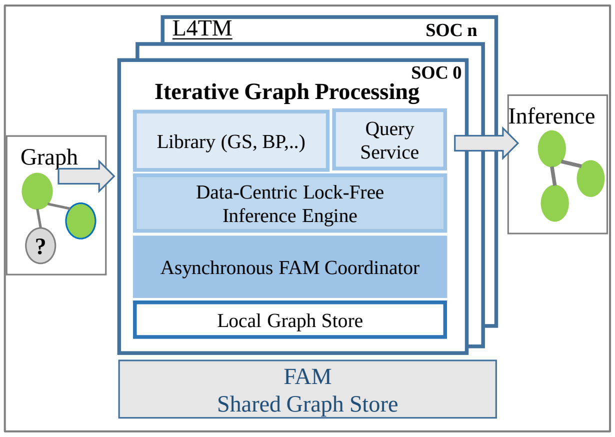


Figure 3. Architecture

iteratively based on the states of the neighboring vertices as well as the metadata associated with the graph until convergence is achieved. The challenge in designing such an engine for The Machine is to take full advantage of the memory bandwidth to enable faster communication of the state variables and thus faster overall computation. Figure 3 shows the architecture of the solution.

Our specific technical contributions are (1) A novel two-level iterative graph processing engine which consists of **a DRAM level lock-free iterative graph processing engine on each SoC** and **an FAM level asynchronous inference engine**, and (2) A **database-inspired** approach to perform optimized iterative computation. We describe them in detail below.

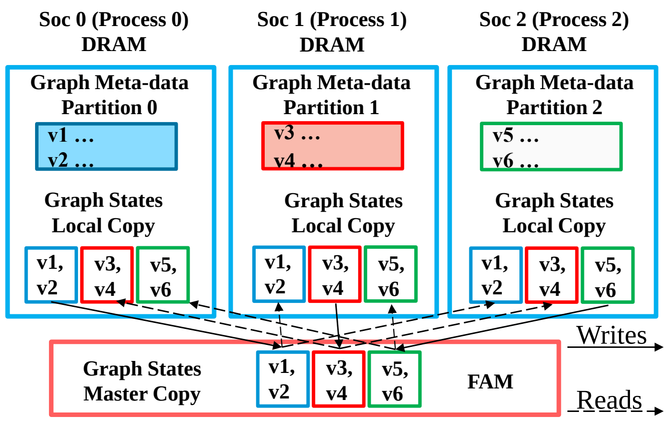


Figure 4 Graph Data Store

Graph Data Store

Figure 4 illustrates how we place and move the data. There are two types of data involved in the iterative computation. The first is the graph topology and the associated metadata such as the factor tables. These are immutable for static graphs and can range in the order of terabytes for a billion node graph. The second is an array of vertex states which is mutable and is in the order of Gigabytes for a Billion node graph. To parallelize the graph processing among various SOCs of The Machine, each SOC is responsible for updating a non-overlapping subarray of the vertex state array. Each SOC keeps only that subset of the graph metadata that is required to update its subarray. The partition size is chosen so that this subset of the metadata fits in DRAM. To update a vertex state the neighboring vertex states are required. Therefore, we replicate the entire array of vertex states on the local DRAM of each SOC. We then exploit the FAM to synchronize the different replicas. We store a master copy of the array of vertex states in the FAM. Periodically, each SOC sends its updates to its subarray to the master copy in the FAM, and then loads the current version of the master copy which contains updates from other SOCs to its DRAM. Since each SOC updates a contiguous subarray and we replicate the entire vertex state array, the writes and reads from each SOC to the master copy are all sequential, and in batch mode.

Data-Centric Lock-Free Inference

We have empirically established that, on a large graph, the probability of thread conflicts is low and the effect of conflict as manifested by dirty data or incorrect computation is mitigated by the iterative nature of the computation such that the final results are unaffected. Hence, we employ lock-free parallelism within each SOC. Multiple threads in each SOC concurrently update and read local vertex states without locks. Since one thread's updates are immediately seen by others, convergence is accelerated. Figures 5 and 6 demonstrate the efficacy of the lock-free approach. Figure 5 plots the number of vertices whose marginal distributions have converged as a function of the

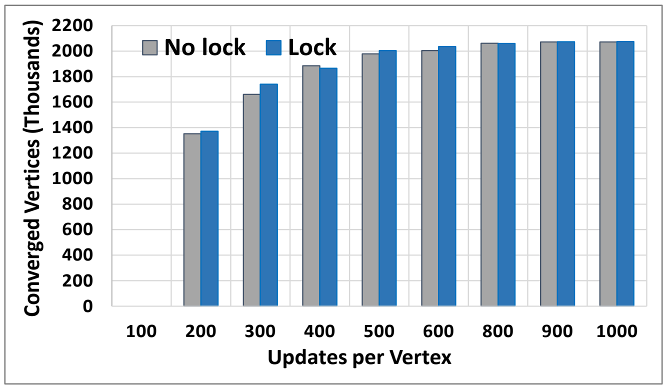


Figure 5: Convergence Rate with Lock and No Lock

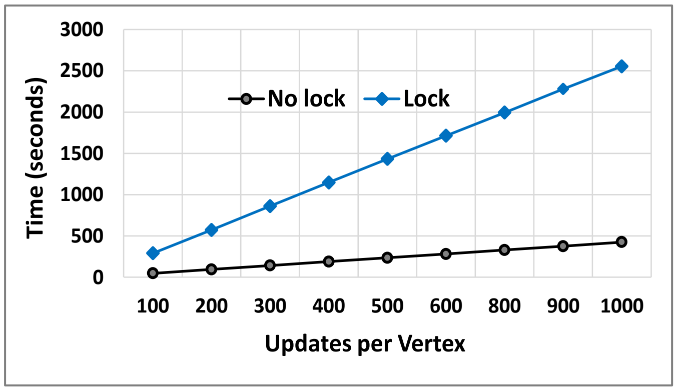


Figure 6 Runtime using 16 threads.

number of updates per vertex of Gibbs sampling for a lock-free implementation and an implementation with locks with 16 threads. The experiment was performed on a 2 million vertex graph that represented domains and hosts in a network. From the figure it is clear that the lock-free version of Gibbs Sampling requires approximately the same number of updates per vertex to attain convergence. However, the total time taken for convergence is much smaller than the implementation with locks. Figure 6 plots the time taken for both implementations against the number of updates. From this plot it is clear that the lock-free implementation is much faster (as much as 5-6X) when compared to the implementation with locks.

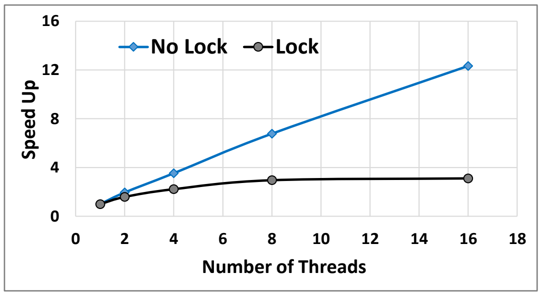


Figure 7 Speed up until convergence.

Finally, in Figure 7 we plot the speed-up obtained over a single-threaded implementation as a function of the number of threads for both implementations. From this plot, we can observe the extent to which locks impede the scaling of the implementations. With 16 threads, the lock-free implementation achieves a speed-up of more than 12x while the implementation with locks plateaus at 3x.

Asynchronous FAM Coordination

Each SOC has a FAM coordinator that periodically updates the master copy of vertex states in FAM with the contents of the SOC's subarray and refreshes the rest of the array in its DRAM based on the master copy. Note that since each subarray is updated by only one SOC, read/write conflicts on the master copy in FAM can only lead to reading stale data, which does not reduce convergence. The frequency at which these updates happen is controlled by a *batch size* parameter which is the number of states updated between two consecutive synchronizations with FAM.

Database inspired optimization

We use a database-inspired approach to update the vertex states. The major operation in updating a vertex state is reading the states of its neighbors. This creates many random accesses on the vertex state array and consumes valuable DRAM bandwidth. To address this, we model this operation as a relational join between the graph metadata and the vertex states shown in Figure 4, and exploit database techniques to optimize it.

Graph inference algorithms are memory-bounded workloads which offer an opportunity to exploit and apply existing relational database techniques to optimize in-memory processing. The translation from a graphical model to a relational model is performed using two categories of tables: edge (E) tables, and vertex (V) tables. An edge (or vertex) table is a table whose size is proportional to the number of edges (vertices) in the graph. Then, the inference computations can be expressed as a joins between these tables. One instance of the relational tables for Gibbs Sampling is shown in Figure 8 (a). The edge table E(xid, fid) records the edges in the graph - for example, the entry (xid, fid) indicates that a vertex with ID xid is connected with a factor with ID fid. The factor table F(fid, ϕ) records the potential function ϕ for factor with ID fid. For GS, we define a table *V* with schema (*xid, v*) where *xid* is the vertex ID, and $v \in D$. To performance the inference, Gibbs Sampling uses a query that aggregates tuples $<x_i, f, \phi, mb(x_i),v>$ for each vertex as a view on x_i. The output of the query is the input to the computation of $p(x_i|mb(x_i))$ and generates a new sample. Then it updates V with x_i new sample *v*.

Similarly, for BP, we define a table B with schema (vid, b), where vid is either a variable ID or a factor ID, and b is the current belief of the corresponding variable (factor). Additionally, we define 2 message tables M_FX with schema (fid, xid, m) and M_XF with schema (xid, fid, m) which contain messages m from factors to variables and from variables to factors respectively. To performance the inference, BP uses two queries: 1) the BP_XF query aggregates the tuple <x, f, b, m> as view on x and updates messages from variables to factors on (outgoing messages) on M_XF; and 2) the BP_FX query aggregates the tuple < f, ϕ, x, b, m> as view on f and update messages from factors to variables (incoming message) on_M_FX. Finally, it updates the beliefs in B(v, b) using the aggregated messages.

E

x_{id}	f_{id}
x_1	**f_1**
x_2	**f_1**
x_2	f_2
x_3	f_2
x_1	**f_3**
x_4	**f_3**

F

x_{id}	
f_1	0
f_2	
f_3	0.8, 0.2, 0.2, 0.4

V

x_{id}	v
x_1	**?**
x_2	M
x_3	B
x_4	M

(a)

M-FX

f_{id}	x_{id}	m
f_1	x_1	$m_{f_1 \to x_1}(M), m_{f_1 \to x_1}(B)$
f_3	x_1	$m_{f_3 \to x_1}(M), m_{f_3 \to x_1}(B)$
…	…	

M-XF

x_{id}	f_{id}	m
x_4	f_3	$m_{x_4 \to f_3}(M), m_{x_4 \to f_3}(B)$
…	…	

B

v_{id}	b
x_1	$b_{x_1}(M), b_{x_1}(B)$
x_2	…
	…
f_1	$b_{f_1}(B,B), b_{f_1}(B,M) b_{f_1}(M,B), b_{f_1}(M,M)$
f_2	…

(b)

Figure 8 Schema of Tables (a) GS and (b) BP.

Optimizing Inference Queries

As described above, the queries follow a type of "join-then-update" pattern. First, they all involve a join between an edge tables and a vertex tables. Then, the result of the inference implies updating the tables to proceed to the next iteration. In order to optimize this "join-then-update" pattern, we consider that the optimization goal is to produce physical plans with minimal cost. The cost is measured by the number of random reads and writes, since they result in potential cache misses. Therefore, ensuring sequential reads for aggregated tuples shows a significant benefit on the overall performance. Similarly, since the algorithms are iterative and some of tables V, B, M_XF, M_FX are updated during the many iterations, minimizing random writes is also very important. We have derived three methods to optimize GS and BP queries described below.

Order Preserved Join-Then-Update. Exploiting the fact that the GS query uses the same order over all computations, we first order the tuples in the edge table so that (1) tuples containing edges related to the same variable are clustered together, and (2) groups are ordered in the same order as the vertex table. During iterations, we perform an index nested-loop join between the edge and vertex table, and then sequentially scan the resulting join results to update the vertex table. In order to optimize the join, we materialize EEF(x_1, x_2, f, ϕ) view which contains all pairs of variables, where each pair contains a variable x_i and a variable x_i in mb(x_i) (see Figure 9). The order on x_1 is preserved and allows a sequential scan to update V. The updated V still preserves the same order on x. This technique avoids sorting the table in each iteration of the Gibbs sampling task.

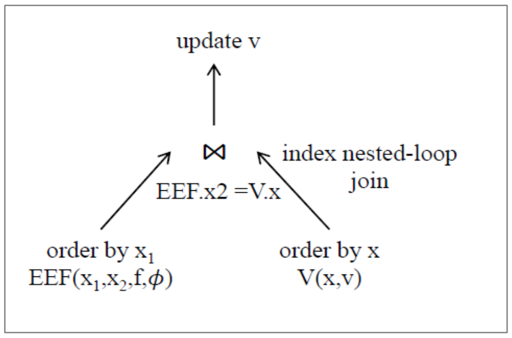


Figure 9. Optimized Join Query for GS

Order Insensitive Join-Then-Update. Since BP queries aggregate and update over two different orders – factors to variables and variables to factors, our solution reduces reads by trading off the cost of more CPU computation. Specifically, to update an incoming message from factor f to variable x, we do not read the outgoing message from x to f. Instead, we compute it on the fly. Computing the outgoing message from x to f only needs to access the belief of x and the current message from f to x. Therefore, it does not need to access the outgoing message tables. As is shown in Figure 10, we first sort both B (f, b) and M XF(x, f, m) on f. Then we join, M XF(x, f, m) with B using an index nested-loop join. The resulting BP XFN (x, f, b, b′, m) table preserves the order on f to update M XF(x, f, m). Furthermore, since tuples in BP XFN are grouped by f, we can also update B(f, b) without any reordering. Additionally, the overall optimization led to use a **unified query** using a **vertex-centric**

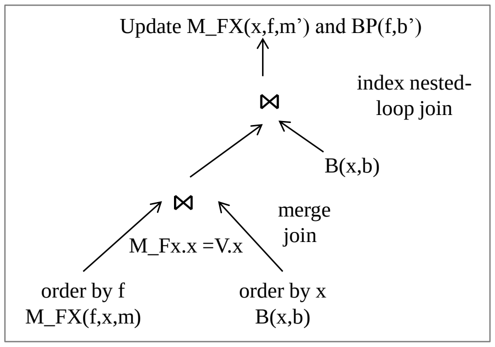


Figure 10. Unified Query of BP

approach for BP(similar to GS) instead of the more natural but expensive edge-centric approach.

Other Optimizations and Final Query Execution Plans.
Another technique to optimize the inference queries is view materialization. On one hand, if we materialize all joins, it significantly reduces random read accesses. On the other hand, this fully materialized view requires a significant number of random writes during iterations. View materialization exploits such tradeoffs. We follow work in [4] or GS to materialize joins differently for sparse and dense graphical models.

IV. Results

We implemented our engine from scratch using C++ and it is available at https://github.hpe.com/labs/LSGi .We evaluated our solution using both real-world and synthetic datasets on two iterative algorithms: Gibbs Sampling (GS) and Belief Propagation (BP)[1]. We describe our results below.

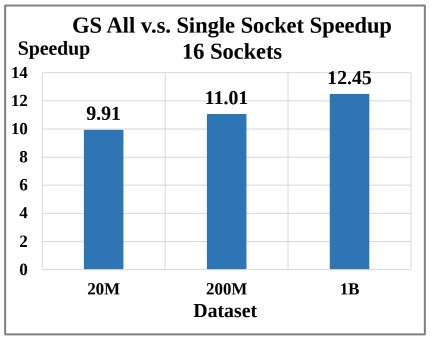
Hardware and System Setup: We evaluated the scalability of our solution on HPE Integrity Superdome X, which has 16 sockets (where each socket is treated like a SOC would be on The Machine), each with 15 CPU cores (Intel(R) Xeon(R) CPU E7-2890 v2 with 2.80 GHz frequency), and 750GB DRAM. The OS running on the server is RedHat 7 Kernel 3.18.0-rc2 x86_64 GNU/Linux with hyper-threading enabled. This OS will be replaced for Linux for The Machine (L4TM) to support FAM storage. As a surrogate of **FAM storage**, we have used **/dev/shm** to store the shared states. /dev/shm is temporary file storage also known as *tmpfs* which is mounted in DRAM instead of persistence memory, e.g. disk. Since Superdome X is a NUMA-aware machine, we run the experiments using *numactl* command to ensure that each process and its threads run and allocate data in the specified SOC. The code is compiled using g++ (GCC) 4.8.3.

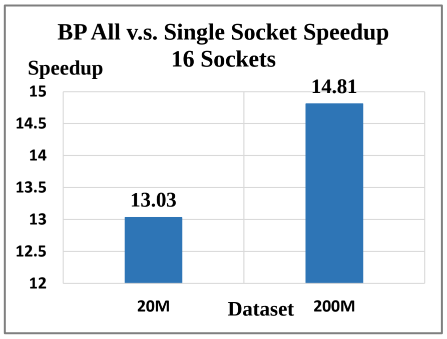
Scalability: Figure 11(a) illustrates the scalability of GS on power-law graphs with 20 Million, 200 Million and 1 Billion vertices with average degree 6 described in table 1.

Dataset	# of Variables (Million)	# of Factors (Million)	Raw File Size (GB)	GS Memory Footprint (16 SOCs)	BP Memory Footprint (16 SOCs)
20M	20	56	4	20 GB	38 GB
200M	200	629	44	200 GB	250 GB
1B	1000	3392	229	1TB	1.3 TB

Table 1: Dataset Settings

As the graph sizes increase, our engine achieved scalability with a 12.6X speedup over a single SOC on the Billion node graph. Figure 11 (b) shows the similar scalability results for BP. We have observed that the synchronization time consumed by the "FAM coordinator" increases as a result of the bottleneck effect caused by all process accessing /dev/shm and flushing cache lines. Figure 11(c) shows that synchronization ratio between 4, 8 and 16





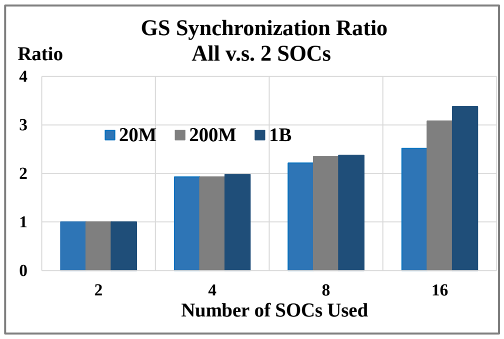
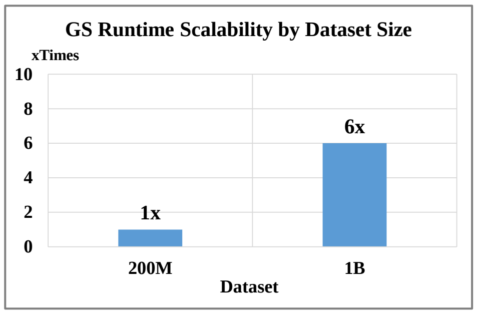


Figure 11. (a) GS Scalability (b) BP Scalability (c) GS Shared States Synchronization, Batch size= 1 Iteration

SOCs v.s. 2 SOCs with a batch size equal to one iteration. The ratio is almost proportional to the number of SOCs involved. We also tested the scalability of our solution with respect to dataset sizes and observed that when the graph was 5X larger, the runtime increased by 6X, which is almost linear as is shown in Figure 12 (a). Similarly, the absolute synchronization time increased almost linear to the graph size.



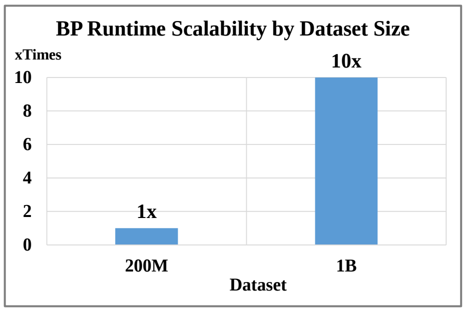


Figure 12. (a) GS Runtime Scalability (b) BP Runtime Scalability

Comparison with GraphLab: Figure 13 shows how much faster our engine was when compared to GraphLab, a state-of-the-art graph processing engine[3][5], on the largest graphs GraphLab could handle on the same hardware. Not only can our engine handle graphs 5-50X larger in size, but it also runs 2-3 orders of magnitude faster on graphs of the same size.

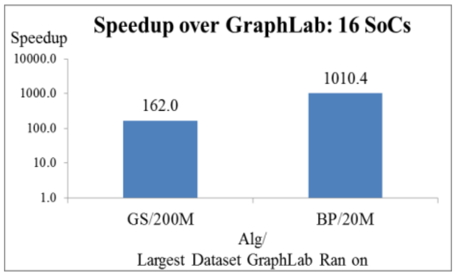


Figure 13. Comparison w GraphLab

Performance Projection on The Machine: Our experiments on the Superdome X used a DRAM region to simulate FAM. The total runtime of our engine can roughly be divided into two parts: in-memory computation and data synchronization with master copy on FAM. To project the performance of our engine on The Machine, we estimated the time taken on The Machine based on the differences in the bandwidth between DRAM and FAM. We project that our engine running GS on The Machine on a 200 Million node graph will be 85 times faster than GraphLab on Superdome X because of the FAM bandwidth change.

V. Related Work

The popularity of graph processing has led to the development of two types of engines. The first focuses on supporting navigational queries and transactions. The second, like our engine, deals with advanced graph analytics algorithms. In this category, the state-of-the art engines are Apache Giraph, GraphLab, and GraphX (Spark graph APIs). We compared to GraphLab since recent benchmarks have shown that it is much faster than both GraphX [5] and Giraph-Pregel [6] on advanced analytics workloads and also because it provides a shared memory implementation as we do.

GraphLab started as an open source project from Carnegie Mello University [3]. The graph processing engine exploits multi-core in-memory and distributed environments introducing a vertex-centric approach along with semantics for parallel scheduling, partitioning, and serialization consistency mechanisms. The framework comprises three main abstractions: the data graph, update function and synchronization mechanism. The data graph is a representation in which each vertex has access to its adjacent edges and vertices. It uses p-vertex cut for graph partitioning minimizing communication across distributed nodes. Each local graph partition maintains ghost copies of the full graph states to have direct visibility and pull updates from remote copies via MPI messages. The update function constitutes the main computation applied to each vertex and its neighborhood e.g page rank, degree count, etc. The execution mechanism supports synchronous and asynchronous models. In the synchronization model, the scheduler has an iteration-barrier ensuring all vertex updates are completed before continue to the next iteration. In the asynchronous processing model, GraphLab avoids iteration barrier by enforcing serialization using a fine-grain locking techniques --edge and vertex consistency to ensure not overlapping on the computation [3][5].

Giraph is an open source which leverage master-worker processing model from Hadoop framework. For graph processing task, Giraph establishes the superstep concept which represents the atomic unit of parallelization. Similar to GraphLab, it exposes a user-defined function which expresses the main computation at vertex granularity in a superstep. Differing from GraphLab, Giraph uses a push model to send messages of new vertex state in a superstep S to adjacent vertices for the superstep S+1.Its passing message approach is also applied to the inter-vertex in a local computation which makes very expensive communication method. The execution model used is bulk synchronization processing (BSP) which led to slow the overall computation waiting for all vertices until their superstep is completed[3].

GraphX is a graph processing framework implemented on top of Apache Spark. Spark is in-memory and distributed framework for big data processing that started as an open source project in Berkeley AMPLlab. The graph is internally represented as vertices and edges using collections of Spark Resilient Distributed Datasets (RDD) along with triples as vertex-edge- vertex and indexes to enable efficient lookups. The graph processing is based on Bulk Parallel Processing (BSP) model and exploits the features already defined for the RDD such as in memory caching, 2D hash partitioning and compressed sparse row. Differing from GraphLab and Giraph, Graphx has adopted edge-centric approach which allows to expand the parallelization by edge instead of vertices. One of the main differentiator of GraphX is the list of optimized relational operators like join, merge, map or reduce which facilitates dynamic graph manipulation, e.g. graph slice, build and transform. GraphX goal is focused on giving flexibility for the analytics pipeline tackling the restriction of GraphLab and Giraph which only operates over static graphs.

Our work has enhanced GraphLab's vertex-centric and p-vertex cut abstractions by featuring our vertex-factor based structure which is optimized for sequential reads and random writes. Our parallel execution model is also aligned to the graph partition ensuring sequential access to vertex neighborhood for local computation. Instead of distributed graph partitions based on p-vertex cut, our engine stored global shared states on FAM avoiding message passing among The Machine Computing nodes. Our research has proven that for large graphs which follow data sparsity property do not require fine grain looking techniques, thus our engine executes asynchronously that demonstrated faster computation approach.

VI. Current status and next steps

We have currently implemented our engine on the HPE Integrity Superdome X, and we have validated for correctness on the Memory Fabric Testbed (MFT) using several emulators and simulators. Via Linux for The Machine(L4TM), we leverage the *libpmem* library and the Librarian File System package on L4TM We are testing our code in The Machine as Simulator (TMAS) and Quality Fabric Environment(QFE) to complete our implementation for The Machine. Our engine is an MFT committed demo slated to be shown at MFT launch in December 2016.

VII. Acknowledgements

We thank our colleagues Brad Morrey, Terence Kelly, Jun Li, Hideaki Kumura, Robert Schreiber and Ram Swaminathan for their support understanding memory mappings, *libpmem* library interfaces, CPU optimization techniques and management support. Thanks to Qiong Cai who developed the model to estimate performance on The Machine based on performance on Superdome X statistics. Thanks to the L4TM and TMAS Team for all the support provided during the test and integration for MFT.

References

[1] *FactorBase: A Data-Centric Approach to Large Scale Probabilistic Inference on NUMA Machines.* F. Chen *et al.* HP Tech. Report 2015

[2] *Graphx: A resilient distributed graph system on spark*. R. Xin *et al*. GRADES, 2013.

[3] *On Characterizing the Performance of Distributed Graph Computation Platforms*. A. Barnawi *et al*. PRFM CHARCS & BM. Springer, 2014.

[4] *Towards high-throughput Gibbs sampling at scale:A study across storage managers,*C. Zhang and C. R´e. SIGMOD-13.

[5] "*Distributed GraphLab: a framework for machine learning and data mining in the cloud*", Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012.. *Proc. VLDB Endow.* 5, 8 (April 2012), 716-727.
[6]" *Unifying data parallel and graph-parallel analytics*", R. Xin, D. Crankshaw, A. Dave, J. Gonzalez, M. Franklin, and I. Stoica. GraphX. arXiv:1402.2394, 2014