UDA-GIST: An In-database Framework to Unify Data-Parallel and State-Parallel Analytics

Kun Li, Daisy Zhe Wang, Alin Dobra, Christopher Dudley
kli@cise.ufl.edu, daisyw@cise.ufl.edu, adobra@cise.ufl.edu, cdudley@cise.ufl.edu
Department of Computer and Information Science and Engineering, University of Florida

ABSTRACT

Enterprise applications need sophisticated in-database analytics in addition to traditional online analytical processing from a database. To meet customers' pressing demands, database vendors have been pushing advanced analytical techniques into databases. Most major DBMSes offer User-Defined Aggregate (UDA), a data-driven operator, to implement many of the analytical techniques in parallel. However, UDAs can not be used to implement statistical algorithms such as Markov chain Monte Carlo (MCMC), where most of the work is performed by iterative transitions over a large state that can not be naively partitioned due to data dependency. Typically, this type of statistical algorithm requires pre-processing to setup the large state in the first place and demands post-processing after the statistical inference. This paper presents General Iterative State Transition (GIST), a new database operator for parallel iterative state transitions over large states. GIST receives a state constructed by a UDA, and then performs rounds of transitions on the state until it converges. A final UDA performs post-processing and result extraction. We argue that the combination of UDA and GIST (UDA-GIST) unifies data-parallel and state-parallel processing in a single system, thus significantly extending the analytical capabilities of DBMSes. We exemplify the framework through two high-profile applications: cross-document coreference and image denoising. We show that the in-database framework allows us to tackle a 27 times larger problem than solved by the state-of-the-art for the first application and achieves 43 times speedup over the state-of-the-art for the second application.

1 Introduction

With the recent boom in Big Data analytics, many applications require large-scale data processing as well as advanced statistical methods such as Random Walk and MCMC algorithms. Connecting tools for data processing (e.g., DBMSes) and tools for large-scale machine learning (i.e., GraphLab [14, 15]) using a system-to-system integration has severe limitations including inefficient data movement between systems, impedance mismatch in data representation and data privacy issues [13, 25]. In the database community, there is a renewed interest in integrating statistical machine learning (SML) algorithms into DBMSes [11]. Such integration allows both SQL-based data processing and statistical data analytics, providing a full spectrum of solutions for data analytics in an integrated system.

Most SML algorithms can be classified into two classes in terms of parallel execution. The first well studied class of SML algorithm requires multiple iterations of the same data. Such SML methods include Linear Regression, K-means and EM algorithms, which can be parallelized within each iteration using naive data partitioning. The overall algorithm can be driven by an outside iteration loop. The parallel implementation of this class of SML algorithm is supported in MADlib [6, 11] and Mahout [16]. Most commercial databases incorporate support for such *data-parallel* SML algorithms in the form of UDAs with iterations in external scripting languages.

A second class of SML algorithm involves pre-processing and constructing a large state with all the data. The state space can not be naively partitioned, because the random variables in the state are correlated with each other. After the state is built, the algorithms involve iterative transitions (e.g., sampling, random walk) over the state space until a global optimization function converges. Such operations are computation intensive without any data flow. After convergence is reached, the state needs to be post-processed and converted into tabular data. We dubbed this class of SML algorithms *state-parallel* algorithms, where the states can be graphs, matrices, arrays or other customized data structures. Examples of this type of SML algorithms include MCMC and belief propagation algorithms.

Several significant attempts have been made towards efficient computation frameworks for SML both in MPP databases such as MADlib [6, 11] and in other parallel and distributed frameworks such as Mahout [16], GraphLab [14, 15] and GraphX [27]. However, no previous work can efficiently support both data-parallel and state-parallel processing in a single system, which is essential for many new applications that applies SML algorithms over large amounts of data. To support such advanced data analytics applications, the UDA-GIST framework developed in this work unifies data-parallel and state-parallel processing by extending existing database frameworks.

Graph-parallel algorithm is a special type of state-parallel algorithm whose state is an immutable graph. Examples of graph-parallel algorithms include inference over large probabilistic graphical models, such as Bayesian Networks [10] and Markov Random Fields [21], where the graph-based state can have hundreds of millions of nodes and billions of edges. While parallel DBMSes and Map-Reduce frameworks can not efficiently express graph-parallel algorithms, other solutions exist such as GraphLab [14, 15] and GraphX [27], both of which have graph-based abstractions. These graph-parallel systems simplify the design and implementation of algorithms over sparse graphs using a high-level abstraction, but they miss the opportunity of using more efficient data structures to represent the state space of a complete/dense graph, a matrix or a dy-

Proceedings of the VLDB Endowment, Vol. 8, No. 5
Copyright 2015 VLDB Endowment 2150-8097/15/01.

namic graph. For example, if the state is a matrix, representing it as a generalized graph can make the state building orders of magnitude slower and hamper the inference significantly due to worse access pattern over a generalized graph. Moreover, GraphLab does not support data-parallel processing for state construction, post-processing, tuples extraction and querying. As shown in the experiments of this paper, it is time consuming to build the state, to post-process and to extract the results, which exceed the inference time. GraphX on the other hand, has a less efficient edge-centric graph representation.

In this paper we ask and positively answer a fundamental question: *Can SML algorithms with large state transition be efficiently integrated into a DBMS to support data analytics applications that require both data-parallel and state-parallel processing?* Such a system would be capable of efficient state construction, statistical inference, post-processing and result extraction.

The main challenge to support efficient and parallel large iterative state transition in-database is the fact that DBMSes are fundamentally data-driven, i.e., computation is tied to the processing of *tuples*. However, iterative state transition based algorithms are computation driven and dissociated from tuples. Supporting such computation needs additional operator abstraction, task scheduling and parallel execution in a DBMS. Secondly, the state has to be represented efficiently inside the DBMS, compatible to the relational data model. Large memory may be required for large states during state transition and new state transition operations have to be efficiently integrated into an existing DBMS.

To solve the first challenge, we introduce an abstraction that generalizes GraphLab API called *Generalized Iterative State Transition* (GIST). GIST requires the specification of an inference algorithm in the form of four abstract data types: 1) the GIST State representing the state space, 2) the Task encoding the state transition task for each iteration, 3) the Scheduler responsible for the generation and scheduling of tasks, and 4) the convergence UDA ensures the stopping condition of the GIST operation gets observed.

We solve the second challenge by efficiently implementing and integrating the GIST operator into a DBMS along with User-Defined Functions (UDFs) [3] and User-Defined Aggregates (UDAs) [26]. The efficient GIST implementation is achieved using the following techniques: 1) asynchronous parallelization of state transition, 2) efficient and flexible state implementation, 3) lock-free scheduler, and 4) code generation. The key of an efficient integration between the non-relational GIST operator and a relational DBMS engine is to use UDAs to build large states from DBMS tuples and to post-process and extract the result tuples from GIST.

The UDA-GIST framework can support a large class of advanced SML-based applications where both data-driven computation and large state transition are required. The specific contributions we make in this paper are:

- We propose a *general iterative state transition* (GIST) operator abstraction for implementing state-parallel SML algorithms. We provide insights and details into how a high performance implementation of GIST can be obtained in a DBMS.
- We explain how a GIST operator implementing the abstraction can be efficiently integrated as a first-class operator in a DBMS. The deep integration of GIST and UDA results in the UDA-GIST framework. We intend the framework to be general for most SML algorithms with support for both data-parallel and state-parallel computation. Compared with GraphLab, the framework trades off implementation complexity for expressiveness and performance. While the application developers may need to implement their own scheduler for synchronization and deadlock resolution, they are given the flexibility to specify their own state representation and parallel execution strategy, which as shown in our experiments can achieve orders-of-magnitude performance gain. The main merits of the UDA-GIST framework are: 1. building state, post-processing and extracting results in parallel. 2. unifying data-parallel and state-parallel computation in a single system. 3. representing states using more compact application-specific data structures. 4. implementing application-specific scheduler for higher-degree of parallel execution. 5. providing an efficient mechanism to detect global/local convergence.
- We exemplify the use of the UDA-GIST abstraction by implementing two high impact SML algorithms and applications: *Metropolis-Hastings algorithm* [2] for cross-document coreference and *loopy belief propagation* [19] for image denoising. We show that two applications can be executed using the extended DBMS execution engine with the UDA-GIST framework. These two applications exemplify the efficiency of the UDA-GIST framework for large classes of state-parallel SML methods such as Markov-chain Monte Carlo and message passing algorithms.
- We show that UDA-GIST framework results in orders-of-magnitude speedup for the two exemplifying applications comparing with state-of-the-art systems. For the first application, using a similar coreference model, features and dataset as described in a recent effort at Google [24], the UDA-GIST system achieves comparable results in terms of accuracy in 10 minutes over a multi-core environment, while the Google system uses a cluster with 100-500 nodes. Results show that this UDA-GIST system can also handle a 27 times larger datset for coreference. For the second application, we show that UDA-GIST outperforms GraphLab's implementation of image denoising with loopy belief propagation by three orders of magnitude for state building and post-processing, and up to 43 times in overall performance.

In the rest of this paper, we first introduce background knowledge of two SML algorithms in Section 2. A system overview is given in Section 3 and the GIST API is presented in Section 4. Sections 5 and 6 showcase two high-profile SML algorithms and applications that involve large-scale state transition and their implementations using GIST and an integrated DBMS system. Finally, we show that the GIST and an efficient integration with a DBMS system result in orders-of-magnitude performance gain in Section 7.

2 Background: Two SML Applications

This section provides background knowledge of two SML applications and the corresponding algorithms, which are implemented using the UDA-GIST framework. The two applications are Metropolis-Hastings algorithm for cross-document coreference and loopy belief propagation for image denoising, both of which require transition over a large state space.

2.1 Metropolis-Hastings for Cross-document Coference

Cross-document coreference [2] (CDC) is the process of grouping the mentions that refer to the same entity, into one entity cluster. Two mentions with different string literals can refer to the same entity. For example, "Addison's disease" and "chronic adrenal insufficiency" refer to the same disease entity. On the other hand, two mentions with the same string literal can refer to different entities. For example "Michael Jordan" can refer to different people entities. While CDC is a very important task, the state-of-the-art coreference model based on probabilistic graphical models is very computation intensive. A recent work by Google Research shows that such model can scale to 1.5 million mentions with hundreds of machines [24].

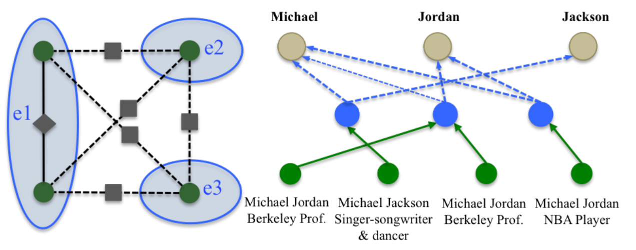

e1
e2
e3
Michael
Jordan
Jackson
Michael Jordan Berkeley Prof.
Michael Jackson Singer-songwriter & dancer
Michael Jordan Berkeley Prof.
Michael Jordan NBA Player

Figure 1: Combined hierarchical model, a combination of pairwise model and hierarchical model. The left graph is a pairwise model consists of 4 mentions (green circles), 3 entities (blue outlined ellipses) and $C_4^2 = 6$ factors. 1 affinity factor is shown with solid line, and 5 repulsion factors with dashed lines. The right graph is the hierarchical model with extra two layers: the super entity and entity. The super entity is identified by a shared token in the top level. The entity represents a cluster of mentions that refer to the same real-world entity.

2.1.1 Pairwise Factor Model for Coreference

A pairwise factor model is a state-of-the-art coreference model [24]. As shown in Figure 1, the model consists of two types of random variables: entities (**E**) and mentions (**M**). Each mention can be assigned to one and only one entity and each entity can have any number of mentions. There is one factor between any pair of mentions m_i and m_j. If the two mentions are in the same entity, the factor is an affinity factor $\psi_a(m_i, m_j)$; otherwise, the factor is a repulsive factor $\psi_r(m_i, m_j)$. Mathematically, we seek the maximum a posteriori (MAP) configuration:

$$\arg\max_{\mathbf{e}} p(\mathbf{e}) = \arg\max_{\mathbf{e}} \sum_{e \in \mathbf{e}} \{ \sum_{m_i, m_j \in e, m_i \neq m_j} \psi_a(m_i, m_j) + \sum_{m_i, m_j \in e, m_i \neq m_j} \psi_r(m_i, m_j) \} \quad (1)$$

Computing the exact **e** is intractable due to the large space of possible configuration. Instead, the state-of-the-art [24] uses Metropolis-Hastings sampling algorithm to compute the MAP configuration.

2.1.2 Hierarchical Model for Coreference

A hierarchical model is presented in a recent paper [24] to scale up CDC which improves the pairwise factor model using a two-level hierarchy of entities in addition to the base mentions: Entities and Super Entities. Given the following concepts:

- $T(m)$: a set of tokens in the string literal of mention m.
- $T(e) = \cup_i T(m_i)$: a union of tokens in the set of mentions that belong to entity e.
- $P(e_s : m \to e_t, T(e_s) \cap T(e_t) \neq \emptyset)$: a proposal to move mention m from source e_s to destination e_t iff the two token set $T(e_s)$ and $T(e_t)$ have at least one common token.

A Super Entity (**SE**) is a `map(key, value)`, where the `key` is a token t and the `value` is a set of entities, whose token set $T(e)$ contains token t. The SE is used as an index to quickly find the target entity in a proposal which has at least one common token with the source entity. The use of SE would increase the effectiveness of the sampling process P to achieve further scalability.

2.2 Loopy Belief Propagation for Image Denoising

The second SML application is image denoising, which is the process of removing noise from an image that is corrupted by additive white Gaussian noise. This is one of the example applications in

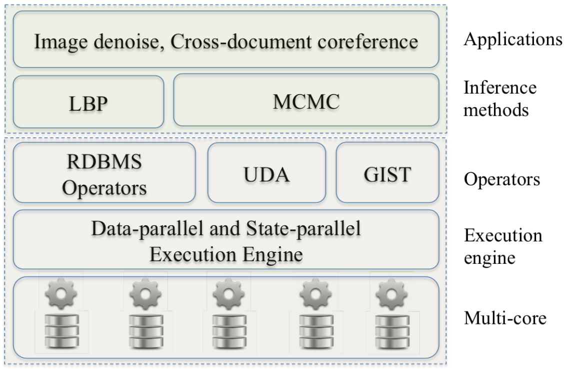


Figure 3: An extended DBMS architecture to support data-parallel and state-parallel analytics.

GraphLab, which uses a 2D grid as the probabilistic graphical model, and 2D mixture as the edge potentials, which enforces neighbouring pixels to have close color. The self potentials are Gaussians centered around the observation. The output is the predicted image. Belief Propagation (BP) [9] is an inference algorithm on Bayesian networks and Markov random fields through message passing. Belief propagation operates on a bipartite factor graph containing nodes corresponding to variables V and factors U, with edges between variables and the factors in which they appear. We can write the joint mass function as:

$$p(\mathbf{x}) = \prod_{u \in U} f_u(\mathbf{x}_u) \quad (2)$$

where $\mathbf{x}_u$ is the vector of neighboring variable nodes to the factor node u. Any Bayesian network or Markov random field can be represented as a factor graph. The algorithm works by passing real valued functions called "messages" along the edges between the nodes. These contain the influence that one variable exerts on another. A message from a variable node v to a factor node u is the product of the messages from all other neighboring factor nodes.

$$\mu_{v \to u}(x_v) = \prod_{u^* \in N(v) \setminus \{u\}} \mu_{u^* \to v}(x_v) \quad (3)$$

where $N(v)$ is the set of neighboring (factor) nodes to v. A message from a factor node u to a variable node v is the product of the factor with messages from all other nodes, marginalized over all variables except x and v:

$$\mu_{u \to v}(x_v) = \sum_{\mathbf{x}'_u : x'_v = x_v} f_u(\mathbf{x}'_u) \prod_{v^* \in N(u) \setminus \{v\}} \mu_{v^* \to u}(x_{v^*}). \quad (4)$$

BP was originally designed for acyclic graphical models. When BP is applied as an approximate inference algorithm over general graphical models, it is called loopy belief propagation (LBP), because graphs typically contain cycles. In practice, LBP converges in many practical applications [12].

3 System Overview

As we explained in the Introduction, state-parallel SML algorithms involve iterative state transitions over a large state space. The execution pipeline for such SML tasks is shown in Figure 2. UDAs are used to construct the in-memory state from the database tables. The GIST takes the state from the UDA and performs iterative state transition over the shared state. The cUDA inside the GIST is used to evaluate the convergence of the state. Finally, the UDA Terminate

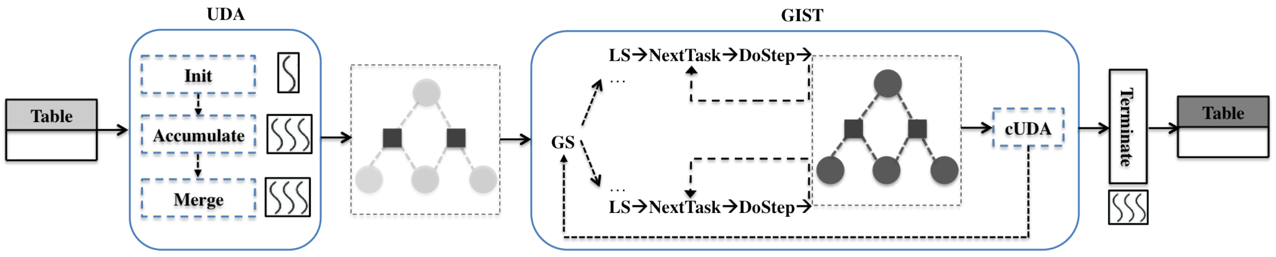


Figure 2: ML Pipeline with large state transition. UDAs are used to construct the in-memory state from the database tables. The GIST takes the state from the UDA and performs computation over the shared state. The cUDA inside the GIST is used to evaluate the convergence of the state. The UDA Terminate function to the right of GIST supports converting the state into relational data.

function to the right of GIST operator supports post-processing and converting the converged final state into relational data.

We design an extended DBMS system architecture that supports the GIST operator and GIST execution engine for iterative state transition over large state space based on a shared-memory paradigm. GIST operators together with data-driven operators in DBMSes such as SQL queries and UDAs can provide efficient and scalable support for a wide spectrum of advanced data analysis pipelines based on SML models and algorithms.

As shown in Figure 3, the GIST operators are implemented as first class citizens, similar as UDAs in an DBMS. UDAs and GIST operators are implemented using two different APIs and are supported by two different execution models. In this paper, the data processing is performed over multi-core machines. Different inference algorithms can be implemented using the GIST and UDA APIs, including loopy belief propagation (LBP) and Markov-chain Monte-Carlo (MCMC) algorithms. Using such inference algorithms, different statistical models can be supported to develop applications such as image denoising and cross-document coreference.

The UDA-GIST framework expands the space of feasible problems on one single multi-core machine and raises the bar on required performance for a complicated distributed system. As an example, our experiments for the coreference application use a 27 times larger dataset than the state-of-the-art in a distributed cluster with 100-500 nodes [24]. One premise of this work is that a single multi-core server is equipped with hundreds of gigabytes of memory, which is sufficiently big to hold the states of most applications. Second, a multi-core server is inexpensive to install, administer and is power efficient. It is hard to acquire and maintain a cluster with hundreds of nodes.

The UDAs follow the traditional API, consisting of three functions: `Init()`, `Accumulate()` and `Merge()`. The `Init` is used to setup the state appropriately before any computation begins. It is similar to a constructor in many high level languages. `Accumulate` takes a tuple as input and adds the tuple to the state that it is maintaining. Tuples can be read from a database or files in the local file system. `Merge` combines the state maintained by two UDA instances of the same type. It requires associativity in nature for computational efficiency, so that states do not need to be merged in any particular order.

4 General Iterative State Transition (GIST)

Compared to the framework proposed by GraphLab [14], GIST API supports more general data structure to represent the state and supports more flexible scheduler for parallel execution. While, by design, GraphLab supports only immutable graph-based data structures, we design GIST to support general data structures to represent large state spaces, including arrays, matrices, and static/dynamic graphs. In addition, we further generalize GraphLab's scheduler in order to allow efficient, parallel execution. In particular, we split the scheduler into a single *global* scheduler (GS) and multiple *local* schedulers (LSs). The GS splits the work into large chunks, one for each local scheduler. The local scheduler manages the chunk and further partitions it into tasks. As we will see, these generalizations allow us to implement inference algorithms more efficiently.

In the rest of the section, we introduce the GIST API and its parallel execution model over a multi-core environment. We then discuss the implementation details of GIST in DataPath and discuss ways to implement GIST in other DBMSes.

4.1 GIST Operator API

Like the treatment of UDA in DBMSes, GIST is an abstract interface that allows the system to execute GIST operators without knowledge of the specifics of the implementation. In this section, we present such an interface and refer to it as the *GIST API*. When designing the API, we have the following desirable properties in mind:

- *Do not restrict the state representation.* Any such restriction limits the amount of *optimization* that can be performed. For example, GraphLab limits the state to generalized graphs, which deteriorate the performance. Graph-based state in Metropolis-Hastings algorithm in CDC forces the graph to be a fully connected graph. The CDC requires full consistency but the full consistency locks all the nodes in the graph–which means no parallelization can be achieved. In the image denoising application, it achieves orders of magnitude speedup by using a matrix state instead of a graph state.
- *Allow fine grained control over the parallel execution.* The applications are free to use their own synchronization primitives. Knowledge of the specifics of the problem allows selection of custom execution strategies. The problem may be better off to use lock-free schedulers and the best effort parallel execution [4, 17], which relaxes the sequential consistency enforced in GraphLab to allow higher degree of parallelism.
- *Allow efficient mechanism to detect state convergence.* Efficient mechanism is needed to detect the convergence in order to make termination decision. We design a convergence evaluation facility to gather statistics in parallel during task execution and make termination decision at the end of each round. This facility enables the computation of global statistics in parallel during inference. This mechanism is not supported either in MADlib or GraphLab.
- *Efficient system integration.* Inference algorithms might require large initial states to be built, then post-processing and extraction of final results from such states. The GIST operator needs to *take over* states built by other means and allows efficient post-

processing and extraction of the result. However, this type of mechanism is missing in GraphLab.

To achieve the above and allow a systematic GIST specification, all GIST operators are represented as a collection of five abstract data types: Task, Local Scheduler, Convergence UDA, GIST State and GIST Terminate. The GIST state will make use of the Task, Local Scheduler, Convergence UDA and GIST Terminate to provide a complete inference model. We discuss each part below starting with the sub-abstractions and finishing with the GIST State.

Task A task represents a single transition that needs to be made on the state, and contains any information necessary to perform this transition. It may be a custom-made class, a class from a pre-made library, or even a basic C++ type. It is the job of the Local Scheduler to know what Tasks it needs to produce and the GIST to know what needs to be done to the state given a certain task. Essentially, the Task allows separation of planning and execution.

Local Scheduler (LS) A LS is responsible for producing the Tasks used to perform state transitions. If the ordering of these Tasks is important, it is up to the LS to produce them in the correct order. Intuitively, the tasks specified by a LS are executed sequentially, but multiple LSs and their tasks may be executed in parallel. It is important to point out that the LSs do not execute tasks, they just specify which tasks should be executed. Effectively, the LSs contain part of the execution plan to be used later by the GIST state. There is no requirement for the LSs to form a task sequence in advance – creating the task sequence on the fly is allowed. A LS has the following public interface:

```
class LocalScheduler {
      bool GetNextTask( Task& );
};
```

The `GetNextTask()` method stores the next task to be run in the location specified by the parameter. If there are no more tasks to be run, the method should return false, and true otherwise. In addition to this public interface, the LS can have one or more constructors that the GIST state is aware of.

Convergence UDA (cUDA) All inference algorithms need to detect convergence in order to make termination decisions. Detecting convergence requires statistics gathering followed by a termination/iteration decision. To allow such a mechanism to be specified, GIST requires a cUDA to be provided. A cUDA is a specialization of the UDA abstraction that is used to determine whether the GIST is done with inference. The cUDA is executed in parallel, much like a regular UDA. One cUDA instance is associated with one LS and gathers local statistics during the tasks execution for the corresponding LS through the use of `Accumulate`. At the end of a round, the cUDA instances are merged using `Merge` to obtain global statistics to allow the termination/iteration decision to be made through the method `ShouldIterate()`. Specifically, the cUDA API is:

```
class cUDA {
  void Init();
  void Accumulate(...);
  void Merge( cUDA& );
  bool ShouldIterate();
};
```

GIST State The GIST state is a class that represents the shared data over which the threads of execution use to perform the inference task. It contain all information that is global to all GIST functionalities and allows execution of the *abstract execution model (AEM)*. The AEM is a declarative execution model that allows the specification of parallelism without a commitment to a specific execution. First, the AEM of GIST specifies that the state transformation proceeds in *rounds*. Convergence is only checked on a round boundary, thus inference can be stopped only at the end of a round. Second, the work to be performed in a round is split into many Tasks, which are grouped into bundles controlled by Local Schedulers (LSs). Tasks in a bundle are executed sequentially, but multiple threads of execution *can be* used to run on independent bundles. The LSs create/administer the Task bundles and provide the next task in a bundle, if any. The partitioning of a round's tasks into bundles is performed by the GIST State abstraction via the `PrepareRound` method and is, in fact, the planning phase of the round. This method plays the role of the *global scheduler*. This method should not perform any work – the user should assume that there is no parallelism during the execution of `PrepareRound`. The system provides a `numParts` hint that indicates a minimum number of LSs that should be created to take full advantage of the system's parallelization. To perform the work specified by each LS, the GIST State abstraction offers the method `DoStep`. This method executes the task provided as the input, i.e. one transformation of the state, and updates the statistics of the provided cUDA. At the end of a round, i.e. when all the tasks specified by all the LSs are executed, the system will merge the cUDA states and determine if further iterations are needed. The execution proceeds either to another round or result extraction.

It is important to point out that this is just an abstract execution model. The execution engine will make the low level decisions on how to break the work into actual parallel threads, how tasks are actually executed by each of the processors, how the work is split between GIST and other tasks that need to be performed, etc. The AEM allows enough flexibility for efficient computation while keeping the model simple.

The GIST constructor may either take constant literal arguments or pre-built states via state passing, and prepares the initial state. Typically, a UDA is used to build the initial state in parallel, and to provide it in a convenient form to the GIST through the constructor.

The `PrepareRound` method produces the LSs and cUDAs for that round and places them into the vector provided. The integer parameter is a hint provided to the GIST for the number of work units that it can use. It is fine to provide more work units than the hint, but providing less will negatively impact the amount of parallelization that can be achieved. The `DoStep` method takes a Task and a cUDA, and performs a single state transition using the information given by the Task. Any information related to the step's effect on the convergence of the GIST is fed to the cUDA.

```
class GIST {
  GIST(...);
  typedef pair<LocalScheduler*, cUDA*> WorkUnit;
  vector<WorkUnit> WorkUnitVector;
  void PrepareRound(WorkUnitVector&, int numParts);
  void DoStep(Task& task, cUDA& agg);
};
```

GIST Terminate The Terminate facility allows for tuples to be post-processed and produced in parallel as long as the results can be broken into discrete fragments that have no effect on one another. The Terminate must have the following interfaces:

```
int GetNumFragments(void);
Iterator* Finalize(int);
bool GetNextResult(Iterator*, Attribute1Type&,...);
```

The GetNumFragments method is called first, and returns the number of fragments the GIST can break the output into. The Finalize method takes an int, which is the ID of the fragment to be produced. The ID is in the range [0, N), where N is the number of fragments returned by GetNumFragments. The Finalize method returns a pointer to an iterator, which is passed to the GetNextResult method to keep track of what tuples still need to be produced for that fragment. The GetNextResult method takes a pointer to an iterator as the first parameter. Attributes of the output tuples should be stored in the locations specified by the parameters and true should be returned if a tuple as successfully produced, and false if there are no more tuples to produce.

4.2 GIST Execution Model

Using the above API, the GIST can be executed as detailed in Algorithms 1 and 2. Algorithm 1 first initializes the GIST state (line 1) and organizes the work into rounds (lines 2-12). For each round, the local schedulers are produced into the list L (line 3) and then for each available CPU, an available local scheduler and cUDA are used to perform one unit of work (line 10). All the work proceeds in parallel until the unit of work terminates. In order to keep track of global work termination for the round, the workOut variable keeps track of the number of work units being worked on. When all work units are done and list L is empty, all the work in this round has finished. The variable gUDA is a global cUDA that is used to determine whether we need more rounds. Notice that round initialization and convergence detection are executed sequentially.

Parallelism of the GIST execution is ensured through parallel calls to PerformWork. As we can see from Algorithm 2, GetNextTask() is used on the local scheduler corresponding to this work unit to generate a task (line 3) and then the work specified by the task is actually executed (line 4). The process is repeated maxWork times or until all the work is performed (lines 2-5). Lines 6-10 detect whether we need more work on this work unit and whether convergence information needs to be incorporated into gUDA (this work unit is exhausted). The reason for the presence of maxWork is to allow the execution engine to *adaptively execute* the work required. If maxWork is selected such that the function PerformWork executes for 10-100ms, there is no need to ensure *load balancing*. The system will adapt to changes in load. This is a technique used extensively in DataPath. In practice, if numCores argument of GIST Execution is selected to be 50% larger than the actual number of cores and maxWork is selected in around 1,000,000, the system will make use of all the available CPU with little scheduling overhead.

Algorithm 1: GIST Execution

```
Require: S_0 GIST Initial State, numCores, maxWork
 1: S← GIST State(S_0)
 2: repeat
 3:     L← S.PrepareRound(numCores)
 4:     workOut ← 0
 5:     gUDA ← Empty cUDA
 6:     repeat
 7:         C ← AvailableCPU
 8:         w ← Head(L)
 9:         workOut ← workOut+1
10:         C.PerformWork(w, maxWork, L, gUDA)
11:     until L.Empty() AND workOut == 0
12: until !gUDA.ShouldIterate()
```

4.2.1 Integration into Datapath+GLADE

We implement GIST as part of the GLADE [22] framework built on top of DataPath [1]. GLADE has a very advanced form of UDA called *Generalized Linear Aggregate* (GLA) that allows large internal state to be constructed in parallel and to be passed around

Algorithm 2: PerformWork

```
Require: w WorkUnit, maxWork, L list<WorkUnit>, S
    GIST state, gUDA
 1: ticks ← 0
 2: repeat
 3:     t ← w.first.GetNextTask()
 4:     S.DoStep(t, w.second)
 5: until ticks>=maxWork OR t≠empty
 6: if ticks=maxWork then
 7:     L.Insert(w);
 8: else
 9:     gUDA.Merge(w.second)
10: end if
11: workOut ← workOut-1
```

to constructors of other abstractions like other GLAs or, in this case, GIST States. The above GIST execution model fits perfectly into DataPath's execution model. We add a waypoint (operator), GIST, to DataPath that implements the above execution model. The user specifies the specific GIST by providing a C++ source file implementing objects with the GIST API. The code generation facility in DataPath is used to generate the actual GIST Operator code around the user provided code. The planning part of the GIST Operator is executed by the DataPath execution engine in the same manner as the UDA, Join, Filter and other operators are executed. Through this integration, GIST operator makes use of the efficient data movement, fast I/O and multi-core execution of DataPath. Since the actual code executed is generated and compiled at runtime, the inference code encoded by GIST is as efficient as hand-written code.

In order to support the large states, which the GIST operator requires as input, we extended the GLA (DataPath+GLADE's UDA mechanism) to allow parallel construction of a single large state (to avoid costly merges) and *pass by STATE* mechanism to allow the state to be efficiently passed to the GIST operator. These modifications were relatively simple in DataPath due to the existing structure of the execution engine.

4.3 Implementing GIST in other DBMSes

In general, adding an operator to a DBMS is a non-trivial task. Both open source and commercial engines have significant complexity and diverse solutions to the design of the database engine. In general, there are two large classes of execution engines: MPP and shared-memory multi-threaded. We indicate how GIST can be incorporated into these two distinct types of engines.

MPP DBMSes, e.g. Greenplum, have a simple execution engine and use *communication* to transfer data between the execution engines. In general, the MPP engines are single-threaded and do not share memory/disk between instances. In such a database engine, it is hard to implement GIST since it requires the global state to be *shared* between all instances. On systems with many CPU cores and large amounts of memory in the master node, this obstacle could be circumvented by launching a program in the *Finalize* function of UDA which simulates the GIST. Such a program would have a dedicated execution engine that is multi-threaded and will perform its own scheduling. The *work_mem* parameter of the master node should be configured to be large enough to hold the entire state. Current implementation of UDA in PostgreSQL/Greenplum can only return a single value and returning a set of records is not supported. The workaround is to concatenate the result records into a single value, then transform the single value into records using the built-in "unnest" UDF. Several limitations can be seen in this naive integration without DBMS source modification. Firstly, it can not take advantage of other slave nodes in the MPP-GIST stage after the state is constructed in the UDA. Secondly, there is no support to convert the GIST state into DBMS relations in parallel. Lastly, converting from GIST internal state to a single value requires an extra copy of memory to store the state.

Shared memory DBMSes, e.g. Oracle and DataPath, usually have sophisticated execution engines that manage many cores/disks. GIST, a shared memory operator, is a natural fit to the shared memory DBMSes as evidenced by the deep integration of GIST to DataPath. To have a genuine integration to this type of DBMS, The UDA should be extended to support passing the constructed state into GIST. The GIST operator needs to make use of the low-level CPU scheduling facilities. Specifically how this can be accomplished depends on the specific DBMS implementation. As a practical integration, the source code of the integration of GIST to DataPath can be found in [8]. To have a shallow integration, the approach is similar as discussed in the integration of GIST to MPP DBMSes.

5 Application I: Cross-document Coreference

The cross-document coreference process involves two distinct stages. The first stage builds the coreference initial state using a UDA. Then a GIST parallel implementation of Metropolis-Hasting algorithm [5] is employed on the initial state until the state has been converged. Figure 4 depicts the pipeline of cross-document coreference.

5.1 GIST Building State in Parallel

As explained in Section 3, the GIST operator relies on a UDA to construct the initial state with data in the mention relation. The UDA builds the initial state required for coreference by accumulating the super entity, entity and mention data structures, and the relationships among them. This is performed via the `Init`, `Accumulate` and `Merge` UDA functions:

Init Build containers for super entities, entities and mentions.

Accumulate The mention relation contains three columns: mention id, the mention string and the surrounding context of the mention. Each `Accumulate` call builds up one mention. Several steps are needed to process a mention. First, the mention string and the mention context are tokenized. Then all the stop words in the mention string are removed. Last, the mention tokens and context tokens are sorted to speed up the calculation of cosine distance at the inference stage. Each `Accumulate` also builds one entity by taking the mention as input since each entity is initialized with one mention at the initial state.

Merge `Merge` simply merges the lists of super entities, entities and mentions.

5.2 GIST Parallel MCMC Sampling

5.2.1 Technical Issues

When implementing an application such as CDC using parallel MCMC sampling, a number of technical difficulties emerge. We briefly discuss them and our solution below.

Parallel random number generator All Monte Carlo methods depend fundamentally on large amounts of pseudo-random numbers (PRN). Most implementations of PRN generators use a global state and random state transitions. Since parallel access to the state creates race conditions – they can result in corruption of the state and the crash of the application – most implementations use a lock to guard the state of the PRN generator. This immediately creates a severe bottleneck in the parallel implementation of MCMC with a drastic decrease in performance. To overcome this problem, we implement a *parallel* version of PRN generation by instantiating a private copy of the PRN for each executing thread. This way, there is no need for locking and the bottleneck is removed.

Deadlock prevention The MCMC based coreference algorithm we use needs to move mentions between entity clusters. This process involves inserting and removing mentions from entity data structures. Since these data structures are not atomic, race conditions can appear that can result in system crashes. The classic solution is to use locks to guard changes to the source and destination entities. Since two locks are needed, deadlocks are possible. To prevent this, we acquire the locks in a set order (based on entity ID).

5.2.2 GIST Parallel MCMC Implementation

The parallel MCMC can be expressed using the GIST with abstract data types: `Task`, `Scheduler`, `cUDA` and `GIST state`.

Task A task in the GIST implementation is a MCMC proposal which contains one source entity and one target entity. Furthermore, the `DoStep` function acquires the locks of the source entity and target entity in order. After obtaining the locks, it picks a random mention from the source entity, then it proposes to move the source mention to the target entity. The task also keeps one reference to its `LS` since the task will consume some computation units, where the outstanding computation units are maintained in its `LS`.

Scheduler The global scheduler assigns the same amount of computation units, which is defined as one time factor computation between two mentions to each `LS`. The number of computation units is stored in variable `numPairs` in the LS. Each LS does Metropolis-Hastings inference until the computation units are consumed for the current iteration.

cUDA The cUDA is used to specify the MCMC inference stopping criteria. Usually, the convergence will be determined by a combination of the following criteria: (a) the maximum number of iterations is reached, (b) the sample acceptance ratio is below threshold, and (c) the difference of F1 between current iteration and last iteration is below threshold. The criterion of maximum number of iterations can be simply implemented by making sure the current iteration is not greater than the maximum number of iterations in the `ShouldIterate()`. To measure the sample acceptance ratio, each `cUDA` will measure the number of accepted proposals and the total number of proposals. Then all the cUDAs will be merged into one state to calculate the overall acceptance ratio. To use the third criterion, the `cUDA` needs to keep track of the last iteration F1 and current iteration F1 in the `cUDA`.

GIST state The GIST state takes the state constructed in the UDA. In addition to the UDA state, GIST state defines the `DoStep` function that transforms the state by working on one task over the state. The `DoStep` function defined in the GIST state is described in algorithm 3. Line 1 generates a random source entity from the entity space. Line 2 picks a random mention in the source entity. Line 3 produces a random token t in the token set of the source entity. Line 4 gets a random target entity in the super entity $S(t)$. Lines 5 and 14 acquire/release the locks of the source entity and target entity in order. Lines 6-13 perform one sample over the state. Line 12 moves all the similar mentions in the source entity to target entity.

6 Application II: Image Denoising

In this application, we implement LBP for image denoising. This is one of the example application implemented in GraphLab. We show that our GIST API can be used to implement LBP and the UDA-GIST interface can be used to support the state construction, inference, post-processing and results extraction.

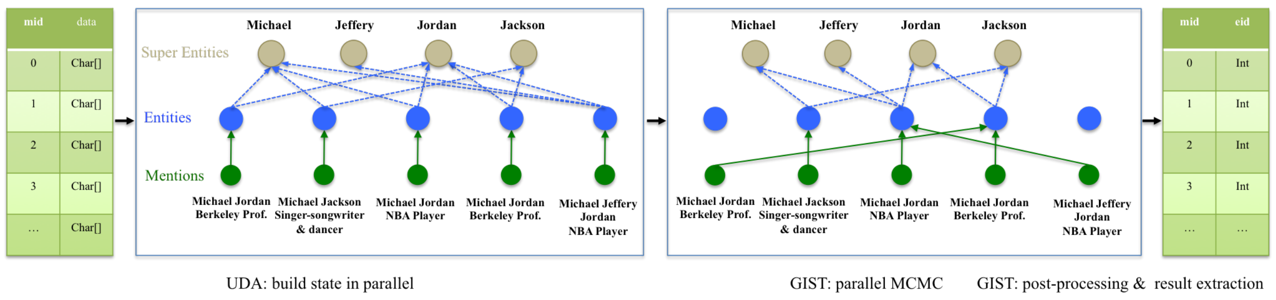

mid
data
Char[]
Super Entities
Entities
Mentions
Michael
Jeffery
Jordan
Jackson
Michael Jordan Berkeley Prof.
Michael Jackson Singer-songwriter & dancer
Michael Jordan NBA Player
Michael Jeffery Jordan NBA Player
eid
Int
UDA: build state in parallel
GIST: parallel MCMC
GIST: post-processing & result extraction

Figure 4: GIST coreference pipeline. A UDA is used to construct the initial hierarchical model where each entity has one mention. The UDA passes the state into GIST, the GIST does parallel MCMC inference over the shared state until the state converges.

Algorithm 3: Metropolis-Hastings `DoStep`

```
Require: Task& task, LocalScheduler& ls
1: $e_s \leftarrow rand(E)$
2: $m_s \leftarrow rand(e_s)$
3: $t = rand(T(e_s))$;
4: $e_t \leftarrow rand(S(t))$
5: lock $e_s, e_t$ in order
6: if $e^{\frac{p(e')}{p(e)}} > rand[0,1]$ then
7:     accepted ← True
8: end if
9: if accepted then
10:     $e_s \leftarrow e_s - m$
11:     $e_t \leftarrow e_t + m$
12:     move the similar mentions in $e_s$ to $e_t$
13: end if
14: unlock $e_s, e_t$ in the reverse order;
```

The LBP process also involves two distinct stages. The first stage is to construct the initial state using a UDA. Then a GIST implementation of the LBP algorithm is employed on the initial state until the state has been converged. Figure 5 depicts the pipeline of LBP inference over the graph state.

6.1 Building GIST State in Parallel

LBP is implemented in GraphLab using a generalized graph data structure since GraphLab uses a generalized graph as its underlying state, where any vertex can connect to any number of vertices either directed or undirected. In many cases, this is not the most efficient representation of the state space.

GIST API can efficiently support different data structures to represent the large state space. We implement this application model using two data structures: graph-based LBP and matrix-based LBP.

6.1.1 Graph-based GIST State

The GraphLab graph state representation is replicated in the GIST graph-based state, where the representation is achieved by maintaining a vector of all the vertices and a vector of all the edges. In addition, two vectors are maintained for each vertex: one vector to keep track of the edge IDs of the incoming edges and the other vector to keep track of the edge IDs of the outgoing edges.

Two UDAs are required to build the initial state. The `vertex-UDA` builds up the graph nodes using the vertex relation. The second UDA `edgeUDA` takes the `vertexUDA` and edge relation as inputs to produce the final graph state. For simplicity, we only shows the implementation of the `vertexUDA`.

Init In `Init`, the `vertexUDA` graph state is set up by allocating space for the vertex vector and two edge vectors for each vertex.

Accumulate Each tuple in the vertex relation is uniquely mapped to one element in the graph state. It achieves massive parallelism since the insertion of tuples into the graph state is done in parallel.

Merge The `Merge` function is left empty since the graph vertex state has been built in the `Accumulate` function.

6.1.2 Matrix-based GIST State

The above graph data structure is too general to exploit the structure in the problem: the pixel neighbours are always the up, down, left and right pixels in the image. A matrix representation captures this connectivity information in a very compact way – the matrix can be thought of as a highly specialized graph. Using a matrix data structure instead of a general graph data structure can significantly reduce the state building time since the state can be pre-allocated and the state initialization can be done in parallel. An `edgeUDA` to construct the edges is not needed in a matrix-based state since the edge connectivity is implicitly stored. Moreover, the performance of the inference algorithms developed on top of the data structure can be sped up. In one run of the vertex program for one vertex, graph-based state requires a sequential access to the in-edge vector and out-edge vector to find the edge IDs and #|edges| random accesses to the global edge vector to modify the edge data. In contrast, in a matrix state, it only requires one sequential scan to the edge data since the edge data for the vertex is maintained locally for each vertex. The detailed UDA implementation is described in the `Init`, `Accumulate` and `Merge` paragraphs.

Init In the `Init` function, a two dimensional matrix state for LBP inference is preallocated by taking the image dimensions.

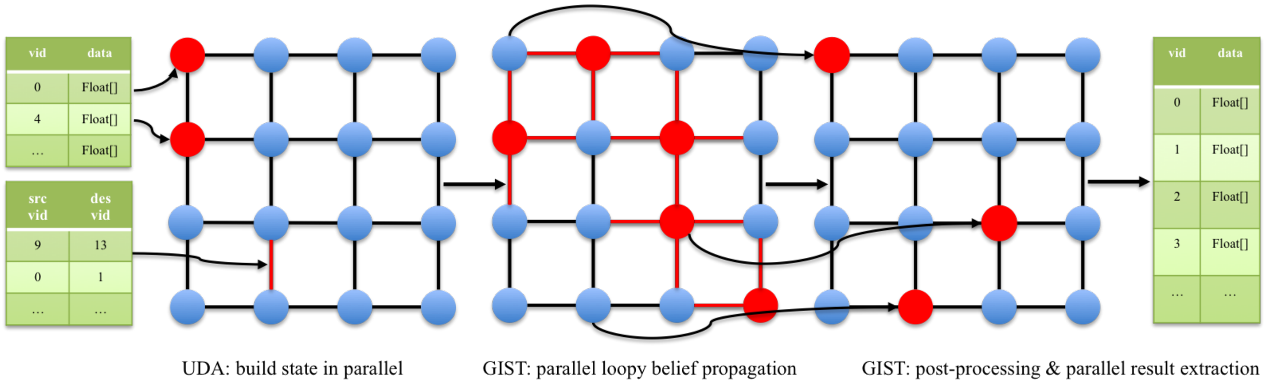
Accumulate Each row in the vertex table stores a vertex id `vid` and the vertex's prior marginal probabilities. A vid uniquely maps to one cell in the matrix state. Thus, it is naively parallel since the initial data assignment for each element in state is independent.

Merge no code is specified in `Merge` since the state has been successfully built with only using the `Init` and `Accumulate`.

6.2 GIST Parallel LBP Implementation

The implementations of GIST parallel LBP based on the graph state and matrix state are almost identical with the help of GIST abstraction. For simplicity, only the matrix-state based LBP implementation is described by demonstrating the implementation of the five abstract data types.

Task One task in the GIST LBP involves polling one vertex from the LocalScheduler (LS) queue and computing the belief based on the messages passed by its neighbors, then computing and sending



vid
data
0
Float[]
4
Float[]
…
Float[]
src vid
des vid
9
13
0
1
…
…
vid
data
0
Float[]
1
Float[]
2
Float[]
3
Float[]
…
…
UDA: build state in parallel
GIST: parallel loopy belief propagation
GIST: post-processing & parallel result extraction

Figure 5: GIST LBP pipeline. A UDA is used to construct the graph state. The UDA passes the state into GIST and the GIST does parallel loopy belief propagation over the shared state until the state converges, which is specified by the convergence UDA.

new messages to its neighbours. Thus the *Task* contains one vertex id. It also keeps a reference to the LS since new task may be dynamically created and pushed back into the current LS.

Scheduler The global scheduler partitions the workloads into local schedulers evenly by assigning the same number of vertices into each LS. If one vertex needs to be resampled, a new task is generated and is pushed back to the end of the LS's task queue. There are no locks needed to extract tasks from LS since no two threads share a LS. In GraphLab, the new task might be inserted into another LS's task queue for the purpose of load balance. Although GraphLab has near-ideal load balance as shown in the experiment but at the cost of significant locking, which hampers the performance tremendously. GIST LBP implementation also relaxes sequential consistency enforced by GraphLab to allow higher degree of parallelism.

Convergence UDA The convergence criterion can be that the maximum number of iterations is reached or the residual of message value is below the terminate bound.

GIST state The GIST state takes the state constructed in the LBP UDA. In addition the UDA state, the GIST state defines the DoStep function that transforms the state by working on one task over the state. The DoStep function defined in the GIST state is described in the Algorithm 4. Line 1 and 2 calculate the coordinates of the current vertex and the current vertex's neighbours. Lines 3-5 calculate the marginal probabilities (beliefs) based on the messages passed by its four neighbours. Lines 6-12 calculate the new messages to its neighbors. If the residual is above threshold, new tasks are dynamically created and added to the task queue.

Algorithm 4: LBP DoStep

```
Require: Task& task, LocalScheduler& $ls$
1:  $V \leftarrow$ {task.vid%dimen, task.vid/dimen}
2:  $N \leftarrow [top, bottom, left, right]$
3:  for all $i \leftarrow 0$ to $4$ do
4:      update local belief $b(V)$ based on message $M_{N[i]\rightarrow V}$
5:  end for
6:  for all $i \leftarrow 0$ to $4$ do
7:      compute message $M_{V\rightarrow N[i]}$
8:      $residual$ = $||M_{V\rightarrow N[i]}$ - $M^{old}_{V\rightarrow N[i]}||$
9:      if $residual < Terminate\ Bound$ then
10:         $ls.addTask(N[i])$
11:     end if
12: end for
```

GIST Terminate After the LBP inference, the data for each vertex needs posterior probability normalization. The fragment output method allows for tuples to be post-processed and produced in parallel. This is an excellent choice for image denoising since it needs to produce large amounts of data as a result and the image can be broken into fragments that have no effect on one another.

7 Experiments

The main goal of the experimental evaluation is to measure the performance of the UDA-GIST framework for the two problems exemplified in this paper: cross-document coreference (CDC) and image denoising. As we will see, when compared to the state-of-the-art, the GIST based solutions scale to problems **27 times** larger for CDC and are up to **43 times** faster for image denoising.

7.1 Experimental Setup

In order to evaluate the C++ GIST implementation of CDC and image denoising, we conduct experiments on various datasets in a multi-core machine with 4 AMD Opteron 6168 running at 1.9GHz processors, 48-core, 256GB of RAM and 76 hard drives connected through 3 RAID controllers. The UDA-GIST framework is implemented in DataPath [1], an open source column-oriented DBMS. The implementation makes use of the GLA , the UDA facility in GLADE [22]. For the image denoising application, we compare the performance with GraphLab. Although we are not able to replicate the same experiment for the CDC in [24] since its source code is not published and the running time is not reported, which make the direct comparison impossible, we show that we are able to tackle a 27 times larger problem than the problem solved in [24]. To the best of our knowledge, GraphLab and Google CDC are written in C++.

7.2 Experimental Datasets

7.2.1 Cross-document Coreference Datasets

The performance and scalability of GIST implementation of cross-document coreference (CDC) is evaluated using the Wikilinks [23] dataset. The Wikilinks dataset contains about 40 millions mentions over 3 millions entities. The surrounding context of mentions is extracted and used as context features in the CDC techniques. In this dataset, each mention has at most 25 left tokens and at most 25 right tokens. The dataset is generated and labelled from the hyperlinks to the Wikipedia page. The anchor texts are *mentions* and the corresponding Wikipedia hyperlinks are *entities*. Any anchor texts which link to the same Wikipedia page refer to the same entity. For our experiments, we extract two datasets Wikilink 1.5 (first

1.5M mentions from the 40M dataset) and Wikilink 40 (all 40M mentions in the dataset) from this Wikilink dataset.

The state-of-the-art [24] evaluates CDC performance using a different Wikilink dataset containing 1.5 million mentions. The exact dataset used and the running time in that paper is not published, thus a direct comparison is not possible. Nevertheless, our version of the Wikilink 1.5 has the same number of mentions and about the same number of entities.

Noticeably, with the exception of [24], no prior work provided experiments with datasets larger than 60,000 mentions and 4000 entities (see [23] for a discussion). The Wikilink 40 dataset is **27 times** larger than the largest experiment reported in the literature.

7.2.2 Image Denoising Datasets

We evaluate the performance with synthetic data generated by a synthetic dataset generator provided by GraphLab [14]. The generator produces a noisy image Figure 8(b) and the corresponding original image Figure 8(a). Loopy belief propagation is applied to reconstruct the image and it produces a predicted image. We use the dataset generator to generate 10 image datasets varying from 4 millions pixels (2000$\times$2000) to 625 million pixels (25000$\times$25000).

7.3 Experimental Results

7.3.1 Cross-document Coreference with Metropolis-Hastings

Methods The performance of GIST coreference is evaluated against the state-of-the-art [24] using a similar dataset, feature set, model and inference method.

- Datasets: Wikilink 1.5 and Wikilink 40
- Feature set: the same feature set as in [24] are used where the similarity of two mentions m and n is defined as :
  $\psi(m, n) = (\cos(m, n) + wTSET\_EQ(m, n))$
  Where $cos(m, n)$ is the cosine distance of the context between mention m and mention n. $TSET\_EQ(m, n)$ is 1 if the mention m and n has the same bag of words disregarding the word order in the mention string, otherwise it returns 0. w is the weight of this feature. In our experiment, we set $w = 0.8$.

Wikilink 1.5 experimental results The performance evaluation results over Wikilink 1.5 are depicted in Figure 6(a). The experiment runs for 20 iterations. Each iteration takes approximately 1 minute. During initialization (iteration 0), each entity is assigned to exactly one mention. The state is built using a UDA in about 10s. The inference starts at iteration 1 after the state is constructed. At iteration 7, the state essentially converges and has precision 0.898, recall 0.919 and F_1 0.907. The F_1 continues to slightly improve up to iteration 20. In the last iteration 20, the measures are precision 0.896, recall 0.929 and F_1 0.912. [24] employs 100-500 machines to inference over a similar dataset.

Wiklink 40 experimental results To evaluate the scalability of our coreference implementation, we use the Wikilink 40 that is 27 times larger than the current state-of-the-art. As the same as the above experiment, each entity is assigned with exactly one mention during initialization. The state building takes approximately 10 minutes. Figure 6(b) depicts the performance of our implementation with this dataset on the experimental machine. We run 20 iterations each with 10^{11} pairwise mention comparisons – each `LocalScheduler` is generating random tasks until the comparisons quota is met. Each iteration takes approximately 1 hour and we can see that the graph converges at iteration 10 with precision 0.79, recall 0.83 and F_1 0.81. This essentially means that, using our solution, within a manageable 10 hour computation in a single system the coreference analysis can be performed on the entire Wikilink dataset.

Discussion A direct comparison to the state-of-the-art [24] is not possible since the dataset used is not published and the time plot in its performance graph is relative time. We believe our results are substantial since our final inference measure is similar as reported in the paper and our experiment evaluation finishes in 10 minutes on a single multi-core machine instead of 100-500 machines for a similar dataset Wikilink 1.5. We are also able to finish the inference in 10 hours for a 27 times larger dataset Wikilink 40. The speedup can be seen as follows: [24] uses MapReduce to distribute the entities among machines. After sampling the subgraphs, the subgraphs need to be shuffled and even reconstructed between machines, which suffers I/O bottleneck. However, we use one single machine with enough memory to store the whole graph and maintain an in-memory super entity structure to speed up the MCMC sampling.

7.3.2 Image Denoising with Loopy Belief Propagation

We evaluate the performance of the three approaches: GraphLab LBP, GIST matrix-based LBP and GIST graph-based LBP against the 10 image datasets. The analytical pipeline is broken up into three stages: state building, inference and result extraction. Figure 7 provides a detailed performance comparison of the three methods for each of the three stages. Due to the more compact representation (no explicit representation of edges), only the matrix-based GIST implementation can build the state with 400 millions and 625 millions vertices image dataset. We are able to perform experiments on images with 4-256 million pixels for all three methods.

Overall performance comparison To sum up all the performance metrics of the three stages, Figure 7(a) describes the overall performance speedup w.r.t. the worst. Clearly, GraphLab performs the worst and its performance speedup against itself is always 1. As shown in the experiment results, graph-based GIST can achieve up to 15 times speedup compared with GraphLab. With a matrix-based abstraction, GIST can achieve up to 43 times speedup compared with GraphLab.

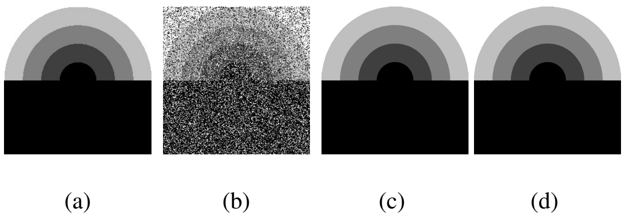


Figure 8: Image denoising with LBP. (a) is the original image and (b) is the corrupted image. (c) is the predicted image by GraphLab and (d) is the predicted image by GIST.

State building In the state building phase, as we can see from Figure 7(b), the graph-based GIST outperforms the GraphLab by up to 16 times speedup with a UDA to construct the graph state in parallel. It is mainly due to the parallel I/O in the DBMS where each UDA instance loads one chunk of the vertex and edge data into memory and the final graph is merged together in the merge function of UDA. GraphLab sequentially constructs the graph state without parallelism as suggested in Figure 9(a), where only one CPU core is used. Matrix-based GIST further improves the state building using a matrix instead of a general graph as the underlying state. The time to build the graph state using a matrix-based GIST is three orders of magnitude faster than the GraphLab. In the matrix-based GIST, a matrix is pre-allocated and UDA instances can pull the data from the disk and fill the matrix independently with massive parallelism.

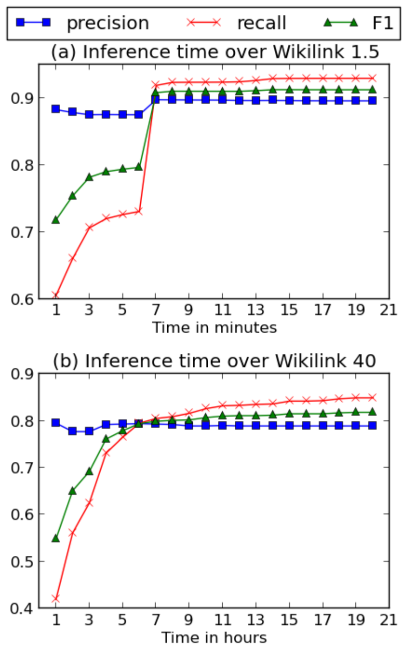


Figure 6: GIST MCMC evaluation over Wikilink 1.5 and Wikilink 40.

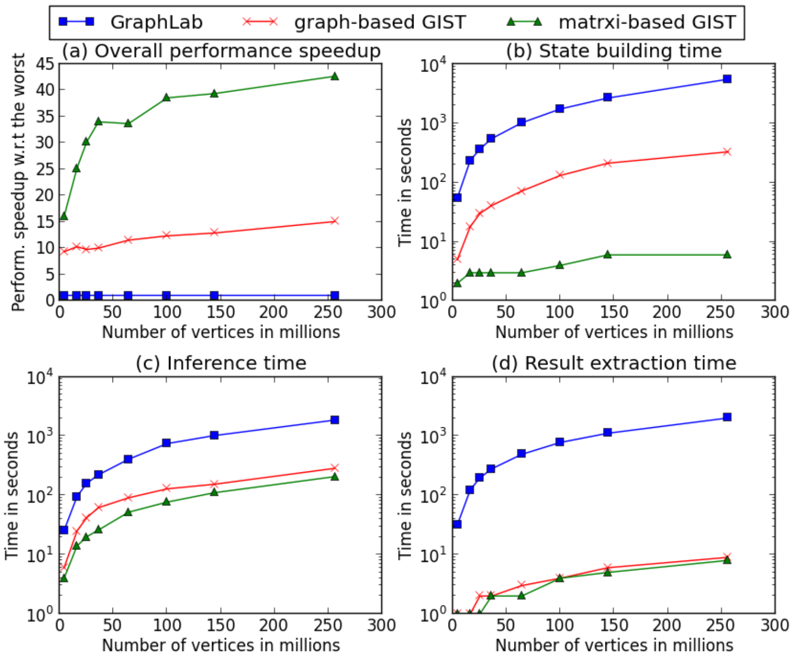


Figure 7: Matrix-based, graph-based GIST LBP (best effort) and GraphLab LBP (sync. engine) performance evaluation over the image datasets.

LBP inference With the identical algorithm implemented in GraphLab, GIST LBP produces the same quality image as GraphLab as shown in Figure 8. The number of vertices sampled in each of the settings is in range [1.45 billion, 1.48 billion]. GraphLab with sync. engine, with async. engine, sweep scheduler and with asyn. engine, fifo_queue scheduler take about 30m, 26m, 24m to converge respectively. Graph-based GIST LBP only takes 4.3m with the lock-free scheduler as discussed in section 6.2. Matrix-based GIST LBP further improves the running time to 3.2m. The 27% performance difference between graph-based and matrix-based GIST is due to the better memory access pattern of the matrix. GraphLab's CPU utilization with sync. engine fluctuates between 6.0 and 45.5 out of 48, where GIST can almost fully utilizes the 48 cores. GraphLab enforces load balancing using the two-choices algorithm in [18] through locking two random task queues. The load balance is not an issue as indicated by the steep decline curve at the end of inference but the cost of locking is significant since the tasks are very lightweight (involving 4 neighbours). Considering data race is rare in a graph with hundreds of millions of nodes for 48 threads, GIST LBP further relaxes sequential consistency to allow high degree of parallelism. Matrix-based GIST LBP with best effort parallel execution (relaxing sequential consistency) converges to the correct point, shown in Figure 8, and improves the running time to 2.5m.

GIST Terminate After the inference, the posterior probability values in vertices of the graph need to be normalized, and then results need to be extracted. GraphLab does not support post-processing and results extraction in parallel since it only has an abstraction for inference. After the parallel inference, GraphLab post-processes each vertex sequentially as shown in the timeline [122, 160] minute of Figure 9(a). The GIST Terminate facility allows for multiple tuples to be post-processed and produced in parallel as depicted in the timeline [3.78, 3.92] minute of Figure 9(b), thus it achieves more than two orders of magnitude speedup over GraphLab.

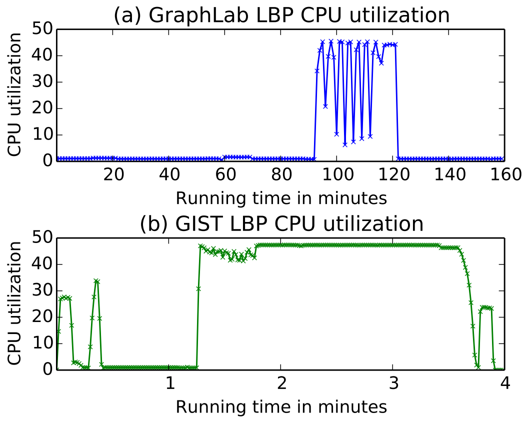


Figure 9: GraphLab LBP (sync. engine) and GIST matrix-based LBP (best effort) CPU utilization evaluation with the image with 256 millions pixels in a single machine with 48 cores.

8 Related Work

Several significant attempts have been made towards efficient computation frameworks for SML in DBMSes such as MADlib [6, 11] and in other parallel and distributed frameworks such as MapReduce [7, 16], GraphLab [14, 15] and GraphX [27].

MADlib [11, 6] integrates data-parallel SML algorithms into DBMSes. By allowing a Python driver for iterations and a UDA to parallelize the computation within each iteration, algorithms like logistic regression, CRF and K-means algorithms are implemented efficiently [13]. However, the data-driven operator, UDA, can not efficiently express state-parallel algorithms. Tuffy [20] attempts an in-database implementation of WalkSAT algorithm over Markov Logic Networks (MLN), but it is too slow for practical use.

Tuffy results in a hybrid architecture where grounding is performed in DBMS and WalkSAT is performed outside of the DBMS. The grounding step over MLN joins the first-order model with data to construct the model, which is the state space consisting of nodes and edges. The sampling step over the MLN is performed outside of a DBMS due to the inefficiency of implementing the state-parallel algorithm using the data-driven execution model. Similar to UDAs, MapReduce excels at expressing data-parallel algorithms but it can not efficiently support state-parallel SML algorithms.

To address the limitations of data-parallel operators to express graph-parallel SML algorithms, GraphLab proposes a computation framework with a graph-based abstraction for graph-parallel algorithms. GraphLab simplifies the design and implementation of SML algorithms , but it can not express SML algorithms whose underlying state are complete graphs, dynamic graphs or more general data structures. As a result, the CDC using the Metropolis-Hastings algorithm where the underlying state is a complete graph, can not be implemented efficiently. Secondly, GraphLab misses the opportunity to exploit the structure of specific problems. For example, the state in the image denoising application can be represented as matrix, a specialized graph. A matrix state brings the opportunity to build the state in parallel with a UDA. It also speeds up the inference due to the better access pattern of matrix. Compared to GraphLab, the UDA-GIST framework further speeds up the performance using lock-free schedulers and best effort parallel execution, which relaxes sequential consistency to allow higher degree of parallelism [4, 17]. Moreover, GraphLab is not integrated with a scalable data processing systems for parallel state construction and parallel result extraction. It is difficult for GraphLab to connect to a DBMS to support a query-driven interface over the data and result due to the impedance mismatch of non-relational world and relational engine.

Pre-processing and post-processing in a graph analytical pipeline are time consuming, which even exceeds the inference time. Motivated by that, GraphX, built on Spark [28], inherits the built-in data-parallel operator to speed up the pre-processing and post-processing. It produces triplets table to represent a graph by joining vertex relation and edge relation. In essence, GraphX is in the same spirit as MapReduce since it is based on a synchronous engine and has data duplication to represent a graph which is different from the graph representation in GraphLab. For inference, GraphX is less efficient than the GraphLab due to the edge-centric graph representation, but it outperforms GraphLab from a end-to-end benchmark, which consists of pre-processing, inference and post-processing [27].

9 Conclusion

In this paper we introduced the GIST operator to implement state-parallel SML algorithms. We presented the UDA-GIST, an in-database framework, to unify data-parallel and state-parallel analytics in a single system with a systematic integration of GIST into a DBMS. It bridges the gap between the relation and state worlds and supports applications that require both data-parallel and state-parallel computation. Since UDA-GIST can consume and produce relational data, complex pipelines involving traditional DBMS operators and SML methods can be pipelined and stacked to support extremely complex computation pipeline. We exemplified the use of GIST abstraction through two high impact machine learning algorithms and showed thorough experimental evaluation that the DBMS UDA-GIST can outperform the state-of-the-art by orders of magnitude.

Acknowledgments

This work was partially supported by generous gifts from Google, Pivotal, EMC/Greeplum, NSF under Grant No. 1144985 and DARPA under FA8750-12-2-0348-2 (DEFT/CUBISM).

10 References

[1] S. Arumugam, A. Dobra, C. M. Jermaine, N. Pansare, and L. L. Perez. The datapath system: a data-centric analytic processing engine for large data warehouses. In A. K. Elmagarmid and D. Agrawal, editors, *SIGMOD Conference*, pages 519–530. ACM, 2010.

[2] A. Bagga and B. Baldwin. Entity-based cross-document coreferencing using the vector space model. In C. Boitet and P. Whitelock, editors, *COLING-ACL*, pages 79–85. Morgan Kaufmann Publishers / ACL, 1998.

[3] T. Bain, L. Davidson, R. Dewson, and C. Hawkins. User defined functions. In *SQL Server 2000 Stored Procedures Handbook*, pages 178–195. Springer, 2003.

[4] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. *SIGPLAN Not.*, 46(8):35–46, Feb. 2011.

[5] S. Chib and E. Greenberg. Understanding the metropolis-hastings algorithm. *The American Statistician*, 49(4):327–335, 1995.

[6] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. Mad skills: New analysis practices for big data. *PVLDB*, 2(2):1481–1492, 2009.

[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150. USENIX Association, 2004.

[8] A. Dobra. Datapath: High-performance database engine, June 2011.

[9] P. F. Felzenszwalb and D. P. Huttenlocher. Efficient belief propagation for early vision. In *CVPR (1)*, pages 261–268, 2004.

[10] N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian network classifiers. *Machine learning*, 29(2-3):131–163, 1997.

[11] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The madlib analytics library or mad skills, the sql. *CoRR*, abs/1208.4165, 2012.

[12] A. T. Ihler, J. Iii, and A. S. Willsky. Loopy belief propagation: Convergence and effects of message errors. In *Journal of Machine Learning Research*, pages 905–936, 2005.

[13] K. Li, C. Grant, D. Z. Wang, S. Khatri, and G. Chitouras. Gptext: Greenplum parallel statistical text analysis framework. In *Proceedings of the Second Workshop on Data Analytics in the Cloud*, DanaC '13, pages 31–35, New York, NY, USA, 2013. ACM.

[14] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. *CoRR*, abs/1006.4990, 2010.

[15] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

[16] A. Mahout. Scalable machine-learning and data-mining library. *available at mahout. apache. org*.

[17] J. Meng, S. Chakradhar, and A. R. Best-effort parallel execution framework for recognition and mining applications. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12, May 2009.

[18] M. Mitzenmacher. The power of two choices in randomized load balancing. *Parallel and Distributed Systems, IEEE Transactions on*, 12(10):1094–1104, 2001.

[19] K. P. Murphy, Y. Weiss, and M. I. Jordan. Loopy belief propagation for approximate inference: An empirical study. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pages 467–475. Morgan Kaufmann Publishers Inc., 1999.

[20] F. Niu, C. Ré, A. Doan, and J. Shavlik. Tuffy: scaling up statistical inference in markov logic networks using an rdbms. *Proceedings of the VLDB Endowment*, 4(6):373–384, 2011.

[21] Y. A. Rozanov. *Markov random fields*. Springer, 1982.

[22] F. Rusu and A. Dobra. Glade: a scalable framework for efficient analytics. *Operating Systems Review*, 46(1):12–18, 2012.

[23] S. Singh, A. Subramanya, F. Pereira, and A. McCallum. Wikilinks: A large-scale cross-document coreference corpus labeled via links to Wikipedia. Technical Report UM-CS-2012-015, 2012.

[24] S. Singh, A. Subramanya, F. C. N. Pereira, and A. McCallum. Large-scale cross-document coreference using distributed inference and hierarchical models. In D. Lin, Y. Matsumoto, and R. Mihalcea, editors, *ACL*, pages 793–803. The Association for Computer Linguistics, 2011.

[25] D. Z. Wang, Y. Chen, C. Grant, and K. Li. Efficient in-database analytics with graphical models. *IEEE Data Engineering Bulletin*, 2014.

[26] H. Wang and C. Zaniolo. User defined aggregates in object-relational systems. In *Data Engineering, 2000. Proceedings. 16th International Conference on*, pages 135–144, 2000.

[27] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[28] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.