Asynchronous Large-Scale Graph Processing Made Easy

Guozhang Wang, Wenlei Xie, Alan Demers, Johannes Gehrke
Cornell University
Ithaca, NY
{guoz, wenleix, ademers, johannes}@cs.cornell.edu

ABSTRACT

Scaling large iterative graph processing applications through parallel computing is a very important problem. Several graph processing frameworks have been proposed that insulate developers from low-level details of parallel programming. Most of these frameworks are based on the bulk synchronous parallel (BSP) model in order to simplify application development. However, in the BSP model, vertices are processed in fixed rounds, which often leads to slow convergence. Asynchronous executions can significantly accelerate convergence by intelligently ordering vertex updates and incorporating the most recent updates. Unfortunately, asynchronous models do not provide the programming simplicity and scalability advantages of the BSP model.

In this paper, we combine the easy programmability of the BSP model with the high performance of asynchronous execution. We have designed GRACE, a new graph programming platform that separates application logic from execution policies. GRACE provides a synchronous iterative graph programming model for users to easily implement, test, and debug their applications. It also contains a carefully designed and implemented parallel execution engine for both synchronous and user-specified built-in asynchronous execution policies. Our experiments show that asynchronous execution in GRACE can yield convergence rates comparable to fully asynchronous executions, while still achieving the near-linear scalability of a synchronous BSP system.

1. INTRODUCTION

Graphs can capture complex data dependencies, and thus processing of graphs has become a key component in a wide range of applications, such as semi-supervised learning based on random graph walks [20], web search based on link analysis [7, 14], scene reconstruction based on Markov random fields [9] and social community detection based on label propagation [19], to name just a few examples. New applications such as social network analysis or 3D model construction over Internet-scale collections of images have produced graphs of unprecedented size, requiring us to scale graph processing to millions or even billions of vertices. Due to this explosion in graph size, parallel processing is heavily used; for example, Crandall et al. describe the use of a 200-core Hadoop cluster to solve the structure-from-motion problem for constructing 3D models from large unstructured collections of images [9].

Recently, a number of graph processing frameworks have been proposed that allow domain experts to focus on the logic of their applications while the framework takes care of scaling the processing across many cores or machines [8, 11, 15, 23, 24, 28, 37, 38]. Most of these frameworks are based on two common properties of graph processing applications: first, many of these applications proceed iteratively, updating the graph data in rounds until a fixpoint is reached. Second, the computation within each iteration can be performed independently at the vertex level, and thus vertices can be processed individually in parallel. As a result, these graph processing frameworks commonly follow the bulk synchronous parallel (BSP) model of parallel processing [32]. The BSP model organizes computation into synchronous "ticks" (or "supersteps") delimited by a global synchronization barrier. Vertices have local state but there is no shared state; all communication is done at the synchronization barriers by message passing. During a tick, each vertex independently receives messages that were sent by neighbor vertices during the previous tick, uses the received values to update its own state, and finally sends messages to adjacent vertices for processing during the following tick. This model ensures determinism and maximizes data parallelism within ticks, making it easy for users to design, program, test, and deploy parallel implementations of domain specific graph applications while achieving excellent parallel speedup and scaleup.

In contrast to the synchronous execution policy of the BSP model, *asynchronous* execution policies do not have clean tick and independence properties, and generally communicate using shared state instead of—or in addition to—messages. Vertices can be processed in any order using the latest available values. Thus, there is no guarantee of isolation between updates of two vertices: vertices can read their neighbor's states at will during their update procedure. Asynchronous execution has a big advantage for iterative graph processing [6, 22]: We can intelligently order the processing sequence of vertices to significantly accelerate convergence of the computation. Consider finding the shortest path between two vertices as an illustrative example: the shortest distance from the source vertex to the destination vertex can be computed iteratively by updating the distances from the source vertex to all other vertices. In this case, vertices with small distances are most likely to lead to the shortest path, and thus selectively expanding these vertices first can significantly accelerate convergence. This idea is more generally referred as the label-setting method for shortest/cheapest path problems on graphs, with Dijkstra's classical method the most popular algorithm in this category. In addition, in asynchronous execution the most recent state of any vertex can be used directly by the next vertex that

6^{th} *Biennial Conference on Innovative Data Systems Research (CIDR'13)* January 6-9, 2013, Asilomar, California, USA.

is scheduled for processing, instead of only using messages sent during the previous tick as in the BSP model. This can further increase the convergence rate since data updates can be incorporated as soon as they become available. For example, in belief propagation, directly using the most recent updates can significantly improve performance over synchronous update methods that have to wait until the end of each tick [12].

Although asynchronous execution policies can improve the convergence rate for graph processing applications, asynchronous parallel programs are much more difficult to write, debug, and test than synchronous programs. If an asynchronous implementation does not output the expected result, it is difficult to locate the source of the problem: it could be the algorithm itself, a bug in the asynchronous implementation, or simply that the application does not converge to the same fixpoint under synchronous and asynchronous executions. Although several asynchronous graph processing platforms have been proposed which attempt to mitigate this problem by providing some asynchronous programming abstractions, their abstractions still require users to consider low-level concurrency issues [17, 21]. For example in GraphLab, the unit of calculation is a single update task over a vertex [21]. When an update task is scheduled, it computes based on whatever data is available on the vertex itself and possibly its neighbors. But since adjacent vertices can be scheduled simultaneously, users need to worry about read and write conflicts and choose from different consistency levels to avoid such conflicts themselves. In Galois, different processes can iterate over the vertices simultaneously, updating their data in an optimistic parallel manner [17]. Users then need to specify which method calls can safely be interleaved without leading to data races and how the effects of each method call can be undone when conflicts are detected. Such conflicts arise because general asynchronous execution models allow parallel threads to communicate at any time, not just at the tick boundaries. The resulting concurrent execution is highly dependent on process scheduling and is not deterministic. Thus, asynchronous parallel frameworks have to make concurrency issues explicit to the users.

For these reasons, a synchronous iterative model is clearly the programming model of choice due to its simplicity. Users can focus initially on "getting the application right," and they can easily debug their code and reason about program correctness without having to worry about low-level concurrency issues. Then, having gained confidence that their encoded graph application logic is bug-free, users would like to be able to migrate to asynchronous execution for better performance *without reimplementing their applications*; they should just be able to change the underlying execution policy in order to switch between synchronous and asynchronous execution.

Unfortunately, this crucially important development cycle — going from a simple synchronous specification of a graph processing application to a high-performance asynchronous execution — is not supported by existing frameworks. Indeed, it is hard to imagine switching from the message-passing communication style of a synchronous graph program to the shared-variable communication used in an asynchronous one without reimplementing the application. However, in this paper we show such reimplementation is unnecessary: most of the benefit of asynchronous processing can be achieved in a message-passing setting by allowing users to explicitly relax certain constraints imposed on message delivery by the BSP model.

Contributions of this Paper. In this paper, we combine synchronous programming with asynchronous execution for large-scale graph processing by cleanly separating application logic from execution policies. We have designed and implemented a large scale parallel iterative graph processing framework named GRACE, which exposes a synchronous iterative graph programming model to the users while enabling both synchronous and user-specified asynchronous execution policies. Our work makes the following three contributions:

(1) We present GRACE, a general parallel graph processing framework that provides an iterative synchronous programming model for developers. The programming model captures data dependencies using messages passed between neighboring vertices like the BSP model (Section 3).

(2) We describe the parallel runtime of GRACE, which follows the BSP model for executing the coded application. At the same time GRACE allows users to flexibly specify their own (asynchronous) execution policies by explicitly relaxing data dependencies associated with messages in order to achieve fast convergence. By doing so GRACE maintains both fast convergence through customized (asynchronous) execution policies of the application and automatic scalability through the BSP model at run time (Section 4).

(3) We experiment with four large-scale real-world graph processing applications written in a shared-memory prototype implementation of GRACE (Section 5). Our experiments show that even though programs in GRACE are written synchronously, we can achieve convergence rates and performance similar to that of completely general asynchronous execution engines, while still maintaining nearly linear parallel speedup by following the BSP model to minimize concurrency control overheads (Section 6).

We discuss related work in Section 7 and conclude in Section 8. We begin our presentation by introducing iterative graph processing applications in Section 2.

2. ITERATIVE GRAPH PROCESSING

Iterative graph processing applications are computations over graphs that update data in the graph in iterations or *ticks*. During each tick the data in the graph is updated, and the computation terminates after a fixed number of ticks have been executed [9] or the computation has converged [13]. We use the belief propagation algorithm on pairwise Markov random fields (MRFs) as a running example to illustrate the computation patterns of an iterative graph processing application [26].

Running Example: Belief Propagation on Pairwise MRF. The pairwise MRF is a widely used undirected graphical model which can compactly represent complex probability distributions. Consider n discrete random variables $\mathcal{X} = \{X_1, X_2, \cdots, X_n\}$ taking on values $X_i \in \Omega$, where Ω is the sample space.[1] A pairwise MRF is an undirected graph $G(V, E)$ where vertices represent random variables and edges represent dependencies. Each vertex u is associated with the potential function $\phi_u : \Omega \mapsto \mathbb{R}^+$ and each edge $e_{u,v}$ is associated with the potential function $\phi_{u,v} : \Omega \times \Omega \mapsto \mathbb{R}^+$. The joint distribution is proportional to the product of the potential functions:

$$p(x_1, x_2, \cdots, x_n) \propto \prod_{u \in V} \phi_u(x_u) \cdot \prod_{(u,v) \in E} \phi_{u,v}(x_u, x_v)$$

Computing the marginal distribution for a random variable (i.e., a vertex) is the core procedure for many learning and inference tasks in MRF. Belief propagation (BP), which works by repeatedly passing messages over the graph to calculate marginal distributions until the computation converges, is one of the most popular algorithms used for this task [12]. The message $m_{u \to v}(x_v)$ sent

[1] In general, each random variable can have its own sample space. For simplicity of discussion, we assume that all the random variables have the same sample space.

Algorithm 1: Original BP Algorithm

Initialize $b_u^{(0)}$ as ϕ_u for all $u \in V$;
Calculate the message $m_{u\to v}^{(0)}$ using $b_u^{(0)}$ according to Eq. 2 for all $e_{u,v} \in E$;
Initialize $t = 0$;
repeat
  $t = t + 1$;
  foreach $u \in V$ **do**
    Calculate $b_u^{(t)}$ using $m_{w\to u}^{(t-1)}$ according to Eq. 1 ;
    foreach outgoing edge $e_{u,v}$ *of* u **do**
      Calculate $m_{u\to v}^{(t)}$ using $b_u^{(t)}$ according to Eq. 2 ;
    end
  end
until $\forall u \in V, ||b_u^{(t)} - b_u^{(t-1)}|| \leq \epsilon$;

Algorithm 2: Residual BP Algorithm

Initialize b_u^{new} and b_u^{old} as ϕ_u for all $u \in V$;
Initialize $m_{u\to v}^{\text{old}}$ as uniform distribution for all $e_{u,v} \in E$;
Calculate message $m_{u\to v}^{\text{new}}$ using b_u^{new} according to Eq. 2 for all $e_{u,v} \in E$;
repeat
  $u = \arg\max_v(\max_{(w,v)\in E} ||m_{w\to v}^{\text{new}} - m_{w\to v}^{\text{old}}||)$;
  Set b_u^{old} to be b_u^{new} ;
  Calculate b_u^{new} using $m_{w\to u}^{\text{new}}$ according to Eq. 1 ;
  foreach outgoing edge $e_{u,v}$ *of* u **do**
    Set $m_{u\to v}^{\text{old}}$ to be $m_{u\to v}^{\text{new}}$;
    Calculate $m_{u\to v}^{\text{new}}$ using b_u^{new} according to Eq. 2 ;
  end
until $\forall u \in V, ||b_u^{\text{new}} - b_u^{\text{old}}|| \leq \epsilon$;

from u to v is a distribution which encodes the "belief" about the value of X_v from X_u's perspective. Note that since MRFs are undirected graphs, there are two messages $m_{u\to v}(x_v)$ and $m_{v\to u}(x_u)$ for the edge $e_{u,v}$. In each tick, each vertex u first updates its own belief distribution $b_u(x_u)$ according to its incoming messages $m_{w\to u}(x_u)$:

$$b_u(x_u) \propto \phi_u(x_u) \prod_{e_{w,u}\in E} m_{w\to u}(x_u) \tag{1}$$

This distribution indicates the current belief about X_u's value. The message $m_{u\to v}(x_v)$ for its outgoing edge $e_{u,v}$ can then be computed based on its updated belief distribution:

$$m_{u\to v}(x_v) \propto \sum_{x_u\in\Omega} \phi_{u,v}(x_u, x_v) \cdot \frac{b_u(x_u)}{m_{v\to u}(x_u)} \tag{2}$$

Each belief distribution can be represented as a vector, residing in some *belief space* $\mathcal{B} \subset (\mathbb{R}^+)^{|\Omega|}$; we denote all the $|V|$ beliefs as $\mathbf{b} \in \mathcal{B}^{|V|}$. Hence the update procedure of Equation (2) for a vertex v can be viewed as a mapping $f_v : \mathcal{B}^{|V|} \mapsto \mathcal{B}$, which defines the belief of vertex v as a function of the beliefs of all the vertices in the graph (though it actually depends only on the vertices adjacent to v). Let $\mathbf{f}(\mathbf{b}) = (f_{v_1}(\mathbf{b}_1), f_{v_2}(\mathbf{b}_2), \cdots, f_{v_{|V|}}(\mathbf{b}_{|V|}))$, the goal is to find the fixpoint $\mathbf{b}^*$ such that $\mathbf{f}(\mathbf{b}^*) = \mathbf{b}^*$. At the fixpoint, the marginal distribution of any vertex v is the same as its belief. Thus BP can be treated as a way of organizing the "global" computation of marginal beliefs in terms of local computation. For graphs containing cycles, BP is not guaranteed to converge, but it has been applied with extensive empirical success in many applications [25].

In the original BP algorithm, all the belief distributions are updated simultaneously in each tick using the messages sent in the previous tick. Algorithm 1 shows this procedure, which simply updates the belief distribution on each vertex and calculates its outgoing messages. The algorithm terminates when the belief of each vertex u stops changing, in practice when $||b_u^{(t)} - b_u^{(t-1)}|| < \epsilon$ for some small ϵ.

Although the original BP algorithm is simple to understand and implement, it can be very inefficient. One important reason is that it is effectively only using messages from the previous tick. A well-known empirical observation is that when vertices are updated sequentially using the latest available messages from their neighbors, the resulting asynchronous BP will generally approach the fixpoint with fewer updates than the original variant [12]. In addition, those vertices whose incoming messages changed drastically will be more "eager" to update their outgoing messages in order to reach the fixpoint. Therefore, as suggested by Gonzalez et al., a good sequential update ordering is to update the vertex that has the largest "residual," where the residual of a message is defined as the difference between its current value and its last used value, and the residual of a vertex is defined as the maximum residual of its received messages [13]. The resulting residual BP algorithm is illustrated in Algorithm 2, where during each tick the vertex with the largest residual is chosen to be processed and then its outgoing messages are updated.

Comparing Algorithm 1 and 2, we find that their core computational logic is actually the same: they are both based on iterative application of Equations 1 and 2. The only difference lies in how this computational logic is executed. In the original BP algorithm, all vertices simultaneously apply these two equations to update their beliefs and outgoing messages using the messages received from the previous tick. In the residual BP algorithm, however, vertices are updated sequentially using the most recent messages while the order is based on message residuals. The independent nature of the message calculation suggests that the original BP algorithm can be easily parallelized: since in each tick, each vertex only needs to read its incoming messages from the previous tick, the vertex computations will be completely independent as long as all messages from the previous tick are guaranteed to be available at the beginning of each tick. Such a computational pattern can be naturally expressed in the BSP model for parallel processing, where graph vertices are partitioned and distributed to multiple processes for local computation during each tick. These processes do not need to communicate with each other until synchronization at the end of the tick, which is used to make sure every message for the next tick has been computed, sent, and received. Such a parallel program written in the BSP model can automatically avoid non-deterministic access to shared data, thus achieving both programming simplicity and scalability.

On the other hand, in the residual BP algorithm only one vertex at a time is selected and updated based on Equation 1 and 2. In each tick it selects the vertex v with the maximum residual:

$$\arg\max_v \max_{(w,v)\in E} ||m_{w\to v}^{\text{new}} - m_{w\to v}^{\text{old}}||$$

which naively would require $O(|E|)$ time. To find the vertex with maximal residual more efficiently, a priority queue could be employed with the vertex priority defined as the maximum of residuals of the vertex's incoming messages [12]. During each tick, the vertex with the highest priority is selected to update its outgoing messages, which will then update the priorities of the receivers correspondingly. Note that since access to the priority queue has to be

serialized across processes, the resulted residual BP algorithm no longer fits in the BSP model and thus cannot be parallelized easily. Sophisticated concurrency control is required to prevent multiple processes from updating neighboring vertices simultaneously. As a result, despite the great success of the serial residual BP algorithm, researchers have reported poor parallel speedup [13].

The above observations can also be found in many other graph processing applications. For instance, both Bellman-Ford algorithm and Dijkstra's algorithm can be used to solve the shortest path problem with non-negative edge weights. These two algorithms share the same essential computational logic: pick a vertex and update its neighbors' distances based on its own distance. The difference lies only in the mechanisms of selecting such vertices: in Bellman-Ford algorithm, all vertices are processed synchronously in each tick to update their neighbors, whereas in Dijkstra's algorithm only the unvisited vertex with the smallest tentative distance is selected. On a single processor, Dijkstra's algorithm is usually preferred since it can converge faster. However, since the algorithm requires processing only one vertex with the smallest distance at a time, it cannot be parallelized as easily as Bellman-Ford. Similar examples also include the push-relabel algorithm and the relabel-to-front algorithm for the network max-flow problem [16].

In general, although asynchronous algorithms can result in faster convergence and hence better performance, they are (1) more difficult to program than synchronous algorithms, and (2) harder to scale up through parallelism due to their inherent sequential ordering of computation. To avoid these difficulties, graph processing application developers want to start with a simple synchronous implementation first; they can thoroughly debug and test this implementation and try it out on sample data. Then once an application developer has gained confidence that her synchronous implementation is correct, she can carefully switch to an asynchronous execution to reap the benefits of better performance from faster convergence while maintaining similar speedup — as long as the asynchronous execution generates identical (or at least acceptable) results compared to the synchronous implementation. To the best of our knowledge, GRACE is the first effort to make this development cycle a reality.

3. PROGRAMMING MODEL

Unlike most graph query systems or graph databases which tend to apply declarative (SQL or Datalog-like) programming languages to interactively execute graph queries [2, 3, 4], GRACE follows batch-style graph programming frameworks (e.g., PEGASUS [15] and Pregel [23]) to insulate users from low level details by providing a high level representation for graph data and letting users specify an application as a set of individual vertex update procedures. In addition, GRACE lets users explicitly define sparse data dependencies as messages between adjacent vertices. We explain this programming model in detail in this section.

3.1 Graph Model

GRACE encodes application data as a directed graph $G(V, E)$. Each vertex $v \in V$ is assigned a unique identifier and each edge $e_{u \to v} \in E$ is uniquely identified by its source vertex u and destination vertex v. To define an undirected graph such as an MRF, each undirected edge is represented by two edges with opposite directions. Users can define arbitrary attributes on the vertices, whose values are modifiable during the computation. Edges can also have user-defined attributes, but these are read-only and initialized on construction of the edge. We denote the attribute vector associated with the vertex v at tick t by S_v^t and the attribute vector associated with edge $e_{u \to v}$ by $S_{u \to v}$. Since $S_{u \to v}$ will not be changed during the computation, the state of the graph data at tick t is determined by $S^t = (S_{v_1}^t, S_{v_2}^t \cdots, S_{v_{|V|}}^t)$. At each tick t, each vertex v can send at most one message $M_{v \to w}^t$ on each of its outgoing edges. Message attributes can be specified by the users; they are read-only and must be initialized when the message is created.

```
class Vertex {
  Distribution potent; // Predefined potential
  Distribution belief; // Current belief
};

class Edge {
  Distribution potent; // Predefined potential
};

class Msg {
  Distribution belief; // Belief of the receiver
                       // from sender's perspective
};
```

Figure 1: Graph Data for BP

As shown in Figure 1, for Belief Propagation, S_v^t stores vertex v's predefined potential function $\phi_v(x_v)$, as well as the belief distribution $b_v(x_v)$ estimated at tick t. $S_{u \to v}$ stores the predefined potential function $\phi_{u,v}(x_u, x_v)$. Note that the belief distributions are the only modifiable graph data in BP. As for messages, $M_{u \to v}^t$ stores the belief distribution $m_{u \to v}^t$ about the value of X_v from X_u's perspective at tick t.

3.2 Iterative Computation

In GRACE, computation proceeds by updating vertices iteratively based on messages. At each tick, vertices receive messages through all incoming edges, update their local data values, and propagate newly created messages through all outgoing edges. Thus each vertex updates only its own data. This update logic is captured by the Proceed function with the following signature:

List<Message> Proceed(List<Message> msgs)

Whenever this function is triggered for some vertex u, it will be processed by updating the data of vertex u based on the received messages msgs and by returning a set of newly computed outgoing messages, one for each outgoing edge. Note that at the beginning, the vertices must first send messages before receiving any. Therefore the received messages parameter for the first invocation of this function will be empty. When executed synchronously, in every tick each vertex invokes its Proceed function once, and hence expects to receive a message from each of its incoming edges.

As shown in Figure 2, we can easily implement the Proceed function for BP, which updates the belief following Equation 1 (lines 2 - 6) and computes the new message for each outgoing edge based on the updated belief and the edge's potential function (lines 7 - 13); we use `Msg[e]` to denote the message received on edge e. Similarly, `outMsg[e]` is the message sent out on edge e. The `computeMsg` function computes the message based on Equation (2). If a vertex finds its belief does not change much, it will vote for halt (lines 14 - 15). When all the vertices have voted for halt during a tick, the computation terminates. This voting-for-halt mechanism actually distributes the work of checking the termination criterion to the vertices. Therefore at the tick boundary if we observe that all the vertices have voted for halt, we can terminate the computation.

Note that with this synchronous programming model, data dependencies are implicitly encoded in messages: a vertex should only proceed to the next tick when it has received all the incoming messages in the previous tick. As we will discuss in the next

```
 1 List<Msg> Proceed(List<Msg> msgs) {
 2   // Compute new belief from received messages
 3   Distribution newBelief = potent;
 4   for (Msg m in msgs) {
 5     newBelief = times(newBelief, m.belief);
 6   }

 7   // Compute and send out messages
 8   List<Msg> outMsgs(outDegree);
 9   for (Edge e in outgoingEdges) {
10     Distribution msgBelief;
11     msgBelief = divide(newBelief, Msg[e]);
12     outMsg[e] = computeMsg(msgBelief, e.potent);
13   }

14   // Vote to terminate upon convergence
15   if (L1(newBelief, belief) < eps) voteHalt();

16   // Update belief and send out messages
17   belief = newBelief;
18   return outMsgs;
19 }
```

Figure 2: Update logic for BP

section, we can relax this condition in various ways in order to execute graph applications not only synchronously, but also asynchronously.

4. ASYNCHRONOUS EXECUTION IN BSP

Given that our iterative graph applications are written in the BSP model, it is natural to process these applications in the same model by executing them in ticks: within each tick, we process vertices independently in parallel through the user-specified Proceed function; at the end of each tick we introduce a synchronization barrier. A key observation in the design of the GRACE runtime is that asynchronous execution policies can also be implemented with a BSP-style runtime: an underlying BSP model in the platform does not necessarily force synchronous BSP-style execution policies for the application.

4.1 Relaxing Synchrony Properties

In the BSP model, the graph computation is organized into ticks. For synchronous execution, each vertex reads all available messages, computes a new state and sends messages that will be available in the next tick. This satisfies the following two properties:

Isolation. Computation within each tick is performed independently at the vertex level. In other words, within the same tick newly generated messages from any vertices will not be seen by the Proceed functions of other vertices.

Consistency. A vertex should be processed if and only if all its incoming messages have been received. In other words, a vertex' Proceed function should be triggered at the tick when all its received messages from incoming edges have been made available at the beginning of that tick.

By ensuring both isolation and consistency in the BSP model, all vertices are processed independently in each iteration using their received messages from the previous iteration. Hence data dependencies implicitly encoded in messages are respected. In asynchronous execution, however, we can relax isolation or consistency (or both) in order to achieve faster convergence. By relaxing consistency, we allow a vertex to invoke its Proceed function before all the incoming messages have arrived; for example, we can schedule a vertex that has received only a single, but important message. By relaxing isolation, we allow messages generated earlier in an iteration to be seen by later vertex update procedures. Therefore, the invocation order of vertex update procedures can make a substantial difference; for example, we may process vertices with "important" messages early in order to generate their outgoing messages and make these messages visible to other vertices within the same tick, where the message importance is intended to capture the messages contribution to the convergence of the iterative computation. In either case, an edge $e_{u,v}$ could have multiple unread messages if the destination vertex v has not been scheduled for some time or it could have no unread message at all if the source vertex u has not been scheduled since the last time v was processed. Hence we must decide which message on each of the incoming edges should be used when we are going to update the vertex. For example, an option would be to use the latest message when there are "stale" messages or to use the last consumed message if no new message has been received.

By relaxing isolation and/or consistency properties, we can efficiently simulate asynchronous execution while running with a BSP model underneath. Combining different choices about how to relax these properties of the BSP model results in various execution policies, which may result in different convergence rates. We now describe the customizable execution interface of the GRACE runtime which enables this flexibility.

4.2 Customizable Execution Interface

Vertex Scheduling. If users decide to relax the consistency property of the BSP model, they can determine the set of vertices to be processed as well as the order in which these vertices are processed within a tick. In order to support such flexible vertex scheduling, each vertex maintains a dynamic priority value. This value, called the *scheduling priority* of the vertex, can be updated upon receiving a new message. Specifically, whenever a vertex receives a message the execution scheduler will trigger the following function:

```
void OnRecvMsg(Edge e, Message msg)
```

In this function, users can update the scheduling priority of the receiver vertex by aggregating the importance of the received message. Then at the start of each tick, the following function will be triggered:

```
void OnPrepare(List<Vertex> vertices)
```

In OnPrepare, users can access the complete global vertex list and select a subset (or the whole set) of the vertices to be processed for this tick. We call this subset the *scheduled vertices* of the tick. To do this, users can call Schedule on any single vertex to schedule the vertex, or call ScheduleAll with a specified predicate to schedule all vertices that satisfy the predicate. They can also specify the order in which the scheduled vertices are going to be processed; currently, GRACE allows scheduled vertices to be processed following either the order determined by their scheduling priorities or by their vertex ids. For example, users can schedule the set of vertices with high priority values; they can also use the priority just as a boolean indicating whether or not the vertex should be scheduled; they can also simply ignore the scheduling priority and schedule all vertices in order to achieve consistency. The framework does not force them to consider certain prioritized execution policies but provides the flexibility of vertex scheduling through the use of this scheduling priority.

Message Selection. If users decide to relax the isolation property of the BSP model, vertices are no longer restricted to using only messages sent in the previous tick. Users can specify which message to use on each of the vertex's incoming edges when processing the vertex in Proceed. Since every vertex is processed and

```
1 void OnRecvMsg(Edge e, Message msg) {
2   // Do nothing to update priority
3   // since every vertex will be scheduled
4 }

1 Msg OnSelectMsg(Edge e) {
2   return GetPrevRecvdMsg(e);
3 }

1 void OnPrepare(List<Vertex> vertices) {
2   ScheduleAll(Everyone);
3 }
```

Figure 3: Synchronous Original Execution for BP

```
1 void OnRecvMsg(Edge e, Message msg) {
2   Distn lastBelief = GetLastUsedMsg(e).belief;
3   float residual = L1(newBelief, msg.belief);
4   UpdatePriority(GetRecVtx(e), residual, sum);
5 }

1 Msg OnSelectMsg(Edge e) {
2   return GetLastRecvdMsg(e);
3 }

1 void OnPrepare(List<Vertex> vertices) {
2   List<Vertex> samples = Sample(vertices, m);
3   Sort(samples, Vertex.priority, operator >);
4   float threshold = samples[r * m].priority;
5   ScheduleAll(PriorGreaterThan(threshold));
6 }
```

Figure 4: Asynchronous Residual Execution for BP

hence sends messages at tick 0, every edge will receive at least one message during the computation. Users can specify this message selection logic in the following function:

Msg OnSelectMsg(Edge e)

This function will be triggered by the scheduler on each incoming edge of a vertex before the vertex gets processed in the Proceed function. The return message msg will then be put in the parameters of the corresponding edge upon calling Proceed. In the OnSelectMsg function, users can get either 1) the last received message, 2) the selected message in the last Proceed function call, or 3) the most recently received message up to the previous tick. Users have to choose the last option in order to preserve isolation.

In general users may want to get any "unconsumed" messages or "combine" multiple messages into a single message that is passed to the Proceed function, we are aware of this requirement and plan to support this feature in future work. For now we restrict users to choosing messages from only the above three options so that we can effectively garbage collect old messages and hence only maintain a small number of received messages for each edge. Note that during this procedure new messages could arrive simultaneously, and the runtime needs to guarantee that repeated calls to get these messages within a single OnSelectMsg function will return the same message objects.

4.3 Original and Residual BP: An Example

By relaxing the consistency and isolation properties of the BSP model, users can design very flexible execution policies by instantiating the OnRecvMsg, OnSelectMsg and OnPrepare functions. Taking the BP example, if we want to execute the original synchronous BP algorithm, we can implement these three functions as in Figure 3. Since every vertex is going to be scheduled at every tick, OnRecvMsg does not need to do anything for updating

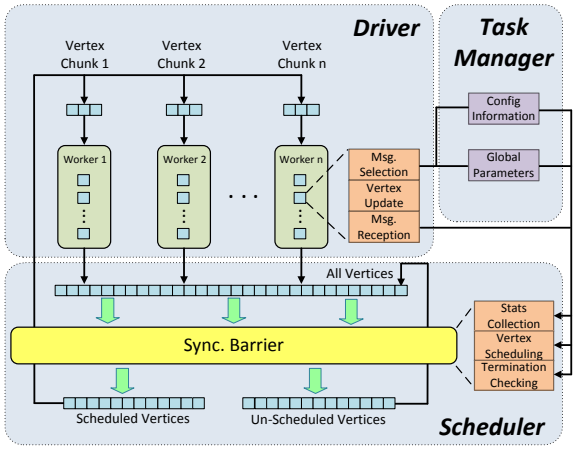

Figure 5: Data flow in GRACE runtime

scheduling priority. In addition, since every edge will receive a message at each tick, OnSelectMsg can return the message received from the previous tick.Finally, in OnPrepare we simply schedule all the vertices. By doing this both consistency and isolation are preserved and we get as a result a synchronous BP program.

If we want to apply the asynchronous residual BP style execution, we need to relax both consistency and isolation. Therefore we can instead implement these three functions as in Figure 4. In this case we use the maximum of the residuals of the incoming messages as the scheduling priority of a vertex. In OnRecvMsg, we first get the belief distribution of the last received message. Next we compute the residual of the newly received message as the L1 difference between its belief distribution with that of the last received message. This residual is then used to update the scheduling priority of the vertex via the sum aggregation function. In OnSelectMsg, we simply return the most recently received message. For OnPrepare, we schedule approximately $r \cdot |V|$ of the vertices with high scheduling priorities by first picking a threshold from a sorted sample of vertices and then calling ScheduleAll to schedule all vertices whose priority values are larger than this threshold.[2]

5. RUNTIME IMPLEMENTATION

We now describe the design and implementation of the parallel processing runtime of GRACE. The goal of the runtime is to efficiently support vertex scheduling and message selection as we discussed in Section 4. Most asynchronous processing frameworks have to make the scheduling decision of which vertex to be processed next. However, because GRACE follows the BSP model, it schedules a set of vertices to be processed for the next tick at the barrier instead of repeatedly schedule one vertex at a time. As we will show below, this allows both fast convergence and high scalability.

5.1 Shared-Memory Architecture

For many large-scale graph processing applications, after appropriate preprocessing, the data necessary for computation con-

[2]To strictly follow the residual BP algorithm, we can only schedule one vertex with the highest scheduling priority for each tick; although this can also be achieved in OnPrepare by first sorting the vertices based on their priorities and then choose only the vertex with the highest priority value, we choose not to demonstrate this execution policy since it is not efficient for parallel processing.

sists of a few hundred gigabytes or less and thus fits into a single shared-memory workstation. In addition, although the BSP model was originally designed for distributed-memory systems, it has also been shown useful for shared-memory multicore systems [33, 36]. As a result, while the GRACE programming abstraction and its customizable execution interface are intended for both shared-memory and distributed memory environments, we decided to build the first prototype of the GRACE runtime on a shared-memory architecture. In the future we plan to extend it to a distributed environment for even larger data sets.

The architecture of the runtime is shown in Figure 5. As in a typical BSP implementation, GRACE executes an application as a group of worker threads, and these threads are coordinated by a driver thread. The runtime also contains a scheduler, which is in charge of scheduling the set of vertices to be processed in each tick. Each run of the application is treated as a *task*, with its config information such as the number of worker threads and task stop criterion stored in the task manager. All vertices, together with their edges and the received messages on each incoming edge, are stored in a global *vertex list* in shared memory. Although messages are logically stored on the edges, in the implementation they are actually located in the data space of the receiving vertex to improve locality.

5.2 Batch Scheduling

Recall that there is a synchronization barrier at the end of each tick, where the user-specified OnPrepare function is triggered. This barrier exists not only for enforcing determinism but also for users to access the global vertex list of the graph. Within this OnPrepare function users can read and write any vertex data. For example, they can collect graph statistics by aggregating over the global vertex list, change global variables and parameters of the application, and finalize the set of scheduled vertices for the next tick. In addition to the user-specified data attributes, the runtime maintains a *scheduling bit* for each vertex. When a vertex is scheduled inside OnPrepare through the Schedule and ScheduleAll function calls, its scheduling bit is set. Thus, at the end of the OnPrepare function, a subset of vertices have their scheduling bits set, indicating that they should be processed in the next tick.

5.3 Iterative Processing

As discussed above, at the end of each tick a set of scheduled vertices is selected. During the next tick these scheduled vertices are assigned to worker threads and processed in parallel. In order to decide the assignment in an efficient and load-balanced way, the driver partition the global vertex list into fixed-size chunks. Those chunks are allocated to worker threads in a round robin manner during the tick. Each worker thread iterates over the vertices of the chunk following either a fixed order or the order specified by vertices' scheduling priority values, selecting and processing vertices whose scheduling bits are set. After a worker thread has finished processing its current chunk, the next free chunk is allocated to it.

When no more free chunks are available to a worker thread, the thread moves on to the tick barrier. Once a worker thread has moved to the tick barrier, there are no more free chunks available to be assigned later to other threads in the current tick. Therefore, the earliest thread to arrive the synchronization barrier will wait for the processing time of a single chunk in the worst case. When every thread has arrived at the tick barrier, we are guaranteed that all the scheduled vertices have been processed. The scheduler then checks if the computation can be stopped and triggers OnPrepare to let users collect statistics and schedule vertices for the next tick.

In the current GRACE runtime, computation can be terminated either after a user-specified number of ticks, or when a fixpoint has been reached. A fixpoint is considered to be reached if the set of scheduled vertices is empty at the beginning of a tick or if all the vertices have voted to halt within a tick. To efficiently check if all the vertices have voted to halt in the tick, each worker thread maintains a bit indicating if all the vertices it has processed so far have voted to halt. At the tick barrier, the runtime only needs to check M bits to see if every vertex has voted to halt, where M is the number of worker threads.

5.4 Vertex Updating

Recall that we use a message passing communication interface to effectively encode the data dependencies for vertices. When the Proceed function is triggered on a vertex, it only needs to access its received messages as well as its local data. Compared to a remote read interface where the Proceed function triggered on a vertex directly accesses the data of the vertex' neighbor vertices, a message passing interface also avoids potential read and write conflicts due to concurrent operations on the same vertex (e.g., a thread updating vertex u is reading u's neighbor vertex v's data while another thread is updating v concurrently). By separating reads and writes, high-overhead synchronization protocols such as logical locking and data replication can usually be eliminated [27, 34].

In the GRACE runtime we implement such a low-overhead concurrency control mechanism for vertex updates as follows. When a vertex is about to be processed, we must select one message on each of its incoming edges as arguments to the Proceed function. Each edge maintains three pointers to received messages: one for the most recently received message; one for the message used for the last call of Proceed; and one for the most recently received message up to the previous tick. Some of these pointers can actually refer to the same message. When any of these message pointers are updated, older messages can be garbage collected. For each incoming edge, the OnSelectMsg function is invoked and the returned message is selected; if OnSelectMsg returns `NULL` the most recently received message is selected by default. The update operations of these message pointers are made atomic and the results of the message pointer de-reference operations are cached such that new message receptions will be logically serialized as either completely before or after the OnSelectMsg function call. Therefore, multiple calls to get a message reference within a single OnSelectMsg function will always return the same result.

After the messages have been selected, the Proceed function will be called to update the vertex with those messages. As described in Section 3, Proceed returns a newly computed message for each of the vertex's outgoing edges, which will in turn trigger the corresponding receiver's OnRecvMsg function. Since during the OnRecvMsg procedure, the receiver vertex's scheduling priority can be updated, and multiple messages reception can trigger the Proceed function simultaneously within a tick, we need to serialize the priority update procedure. The current implementation achieves this with a spin lock. Since the the chance of concurrent Proceed function calls is small and the priority update logic inside Proceed is usually simple, the serialization overhead is usually negligible. Once a scheduled vertex has finished processing, its scheduling bit is reset.

6. EXPERIMENTS

Our experimental evaluation of GRACE had two goals. First, we wanted to demonstrate that by enabling customized execution policies GRACE can achieve convergence rates comparable to state-of-the-art asynchronous frameworks such as GraphLab. Second,

we wanted to demonstrate that GRACE delivers the good parallel speedup of the BSP model even when asynchronous policies are used.

6.1 System

Our shared-memory GRACE prototype is implemented in C++, using the PThreads package. To test the flexibility of specifying customized executions in our runtime, we implemented four different policies: (synchronous) Jacobi (S-J), (asynchronous) GaussSeidel (AS-GS), (asynchronous) Eager (AS-E), and (asynchronous) Prior (AS-P).[3]

Jacobi is the simple synchronized execution policy in which all vertices are scheduled in each tick, and each vertex is updated using the messages sent in the previous tick. It guarantees both consistency and isolation and is the default execution policy in the current GRACE prototype.

The remaining three execution policies are asynchronous; all of them use the most recently received messages. For GaussSeidel, all the vertices are scheduled in each tick, while for Eager and Prior only a subset of the vertices are scheduled in each tick. Specifically, for the Eager execution policy, a vertex is scheduled if its scheduling priority exceeds a user-defined threshold. For the Prior execution policy, only $r \cdot |V|$ of the vertices with the top priorities are scheduled for the next tick, with r a configurable selection ratio. Accurately selecting the top $r \cdot |V|$ vertices in Prior could be done by sorting on the vertex priorities in OnPrepare at the tick barrier, but this would dramatically increase the synchronization cost and reduce speedup. Therefore, instead of selecting exactly the top $r \cdot |V|$ vertices we first sample n of the priority values, sort them to obtain an approximate threshold as the scheduling priority value of the $\lceil r \cdot n \rceil$th vertex, and then schedule all vertices whose scheduling priorities are larger than this threshold.

The implementations of all these execution policies are straightforward, requiring only tens of lines of code in the OnPrepare, OnRecvMsg, and OnSelectMsg functions.

6.2 Workloads

6.2.1 Custom Execution Policies

Recall that the first goal of our experiments was to demonstrate the ability of GRACE to support customized execution policies with performance comparable to existing frameworks. We therefore chose four different graph processing applications that have different preferred execution policies as summarized in Table 1.

Structure from Motion. Our first application is a technique for structure from motion (SfM) to build 3D models from unstructured image collections downloaded from the Internet [9]. The approach is based on a hybrid discrete-continuous optimization procedure which represents a set of photos as a discrete Markov random field (MRF) with geometric constraints between pairs of cameras or between cameras and scene points. Belief Propagation (BP) is used to estimate the camera parameters as the first step of the optimization procedure. We looked into the manually written multi-threaded program from the original paper, and reimplemented its core BP component in GRACE. The MRF graph is formulated in GRACE with each vertex representing a camera, and each edge representing the relative orientation and the translation direction between a pair of cameras. During each tick t, vertex v tries to update its belief S_v^t about its represented camera's absolute location and orientation based on the relative orientations from its neighbors and possibly its priors specified by a geo-referenced coordinate. The manually written program shared by the authors of the SfM paper consists of more than 2300 lines of code for the BP implementation, while its reimplementation in GRACE takes only about 300 lines of code, most of them copied directly from the original program. This is because in GRACE issues of parallelism such as concurrency control and synchronization are abstracted away from the application logic, greatly reducing the programming burden. The execution policy of SfM is simply Jacobi.

Image Restoration. Our second application is image restoration (IR) for photo analysis [31], which is also used in the experimental evaluation of GraphLab [21]. In this application the color of an image is captured by a large pair-wise Markov random field (MRF) with each vertex representing a pixel in the image. Belief propagation is used to compute the expectation of each pixel iteratively based on the observed dirty "pixel" value and binary weights with its neighbors. A qualitative example that shows the effectiveness of the BP algorithm is shown in Figure 6.

In the literature, three different execution policies have been used for this problem, resulting in the following three BP algorithms: the original Synchronous BP, Asynchronous BP, and Residual BP [26, 12]. The execution policy for Synchronous BP is Jacobi, while the Eager policy in GRACE and the FIFO scheduler in GraphLab correspond to the Asynchronous BP algorithm. For the Residual BP algorithm, the Prior policy in GRACE and the priority scheduler in GraphLab are good approximations. Previous studies reported that Residual BP can accelerate the convergence and result in shorter running time compared to other BP algorithms [12, 13].

Topic-Sensitive PageRank. Our third application is topic-sensitive PageRank (TPR) for web search engines [14], where each web page has not one importance score but multiple importance scores corresponding to various topics. At query time, these importance scores are combined based on the topics of the query to form a composite score for pages matching the query. By doing this we capture more accurately the notion of importance with respect to a particular topic. In GRACE, each web page is simply a vertex and each link an edge, with the topic relative importance scores stored in the vertices. Like IR, TPR can use the Jacobi, Eager, and Prior execution policies [38].

Social Community Detection. Our last application is label propagation, a widely used social community detection (SCD) algorithm [19, 29]. Initially, each person has a unique label In each iteration, everyone adopts the label that occurs most frequently among herself and her neighbors, with ties broken at random. The algorithm proceeds in iterations and terminates on convergence, i.e., when the labels stop changing. In GRACE, each vertex represents a person in the social network with her current label, and vertices vote to halt if their labels do not change over an update. Label propagation can be executed either synchronously [19] or asynchronously [29], corresponding to the Jacobi and GaussSeidel execution policies, respectively. Researchers have reported the Label Propagation algorithm converges much faster and can avoid oscillation when using an asynchronous execution policy [29].

6.2.2 Speedup

Recall that the second goal of our experiments was to evaluate the parallel performance of GRACE. We distinguish between *light* and *heavy* iterative graph applications. In heavy applications, such as IR, the computation performed by the Proceed function on a vertex is relatively expensive compared to the time to retrieve the vertex and its associated data from memory. Heavy applica-

[3]The system along with the provided execution policy implementations is available at `http://www.cs.cornell.edu/bigreddata/grace/`

Table 1: Summary of Applications

Application	Comp. Type	Data Set	# Vertices	# Edges	Description
SfM for 3D Model	non-linear	Acropolis	2,961	17,628	Photo Collection downloaded from Flickr
Image Restoration	non-linear	Lenna-Image	262,144	1,046,528	Standard image processing benchmark
Topic PageRank	linear	Web-Google	875,713	5,105,039	Web graph released by Google in 2002
Community Detection	non-numeric	DBLP	967,535	7,049,736	Co-author graph collected in Oct 2011

Figure 6: Qualitative Evaluation of BP Restoration on Lenna Image. Left: noisy (σ = 20). Right: restored

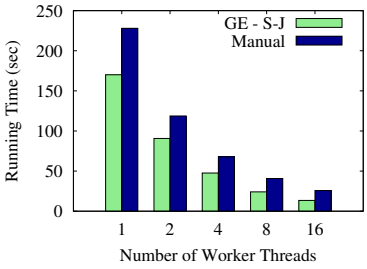

Figure 7: SfM Running Time

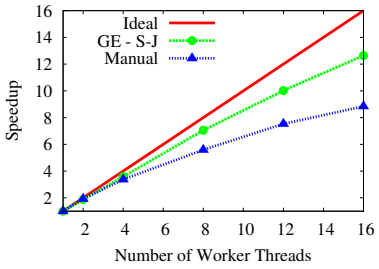

Figure 8: SfM Speedup

tions should exhibit good parallel scaling, as the additional computational power of more cores can be brought to good use. On the other hand, in light applications such as TPR, the computation performed on a vertex is relatively cheap compared to the time to retrieve the vertex's associated data. We anticipate that for light applications access to main memory will quickly become the bottleneck, and thus we will not scale once the memory bandwidth has been reached.

We explore these tradeoffs by investigating the scalability of both IR and TPR to 32 processors. We also add additional computation to the light applications to confirm that it is the memory bottleneck we are encountering.

6.3 Experimental Setup

Machine. We ran all the experiments using a 32-core computer with 8 AMD Opteron 6220 quad-core processors and quad channel 128GB RAM. The computer is configured with 8 4-core NUMA regions. This machine is running CentOS 5.5.

Datasets. Table 1 describes the datasets that we used for our four iterative graph processing applications. For SfM, we used the Acropolis dataset, which consists of about 3 thousand geo-tagged photos and 17 thousand camera-camera relative orientations downloaded from Flickr. For IR we used the "Lenna" test image, a standard benchmark in the image restoration literature [5]. For TPR, we used a web graph released by Google [18], which contains about 880 thousand vertices and 5 million edges, and we use 128 topics in our experiments. For SCD, we use a coauthor graph from DBLP collected in Oct 2011; it has about 1 million vertices and 7 million edges.

We used the GraphLab single node multicore version 2.0 downloaded from `http://graphlab.org/downloads/` on March 13th, 2012 for all the conducted experiments.

6.4 Results for Custom Execution Policies

6.4.1 *Structure from Motion*

First, we used GRACE to implement the discrete-continuous optimization algorithm for SfM problems to demonstrate that by executing per-vertex update procedures in parallel GRACE is able to obtain good data parallelism. We evaluated the performance of the GRACE implementation against the manually written program. Since no ground truth is available for this application, like the manually written program we just executed the application for a fixed number of ticks. Using the Jacobi execution policy, the GRACE implementation is logically equivalent to the manually written program, and thus generates the same orientation outputs. The performance results up to 16 worker threads are shown in Figure 7, and the corresponding speedup results are shown in Figure 8. We can see that the algorithm reimplemented in GRACE has less elapsed time on single CPU, illustrating that GRACE does not add significant overhead. The GRACE implementation also has better multicore speedup, in part because the manually written code estimates the absolute camera poses and the labeling error sequentially, while following the GRACE programming model this functionality is part of the Proceed logic, and hence is parallelized.

6.4.2 *Image Restoration*

To compare the effectiveness of the Prior policy in GRACE with a state of the art asynchronous execution engine, we implemented the image restoration application in both GRACE and GraphLab. We use three different schedulers in GraphLab: the low-overhead FIFO scheduler, the low-overhead splash scheduler, and the higher-overhead prioritized scheduler.

Figure 11 examines the convergence rate of various execution policies; the x-axis is the number of updates made (for GRACE this is just the number of Proceed function calls) and the y-axis is the KL-divergence between the current distribution of each vertex and the ground truth distribution. Here both the priority scheduler of GraphLab and the Prior execution policy of GRACE approximate the classic residual BP algorithm. By comparing with the Jacobi execution policy of GRACE, we can see that Prior yields much faster convergence. For this application such faster convergence indeed yields better performance than Jacobi, which takes about 6.9 hours on a single processor and about 14.5 minutes when using 32 processors.

Figures 9 and 10 compare the running time and the corresponding speedup of GRACE's Prior execution policy with the priority and splash schedulers from GraphLab. As shown in Figure 9, although GraphLab's priority scheduler has the best performance on a single processor, it does not speed up well. The recently introduced ResidualSplash BP was designed specifically for better parallel performance [13]. Implemented in the splash scheduler

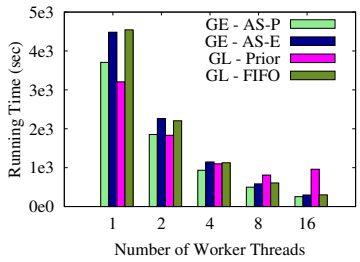

Figure 9: IR Running Time

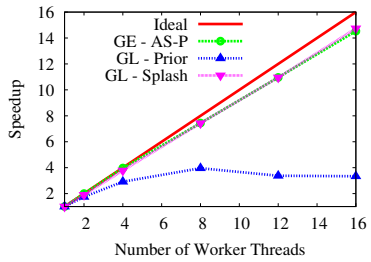

Figure 10: IR Speedup

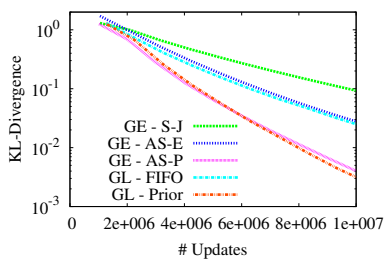

Figure 11: IR Convergence

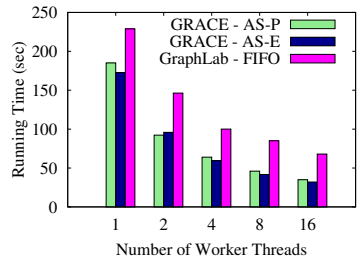

Figure 12: TPR Running Time

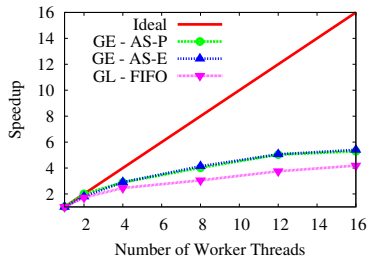

Figure 13: TPR Speedup

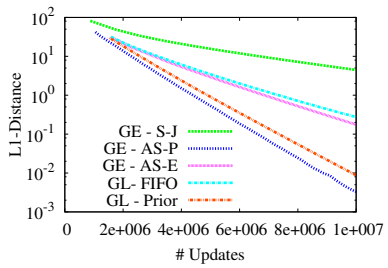

Figure 14: TPR Convergence

of GraphLab, this algorithm significantly outperforms the priority scheduling of GraphLab on multiple processors. As shown in Figure 10, the Prior policy in GRACE can achieve speedup comparable to ResidualSplash BP implemented in GraphLab. This illustrates that by carefully adding asynchronous features into synchronous execution, GRACE can benefit from both fast convergence due to prioritized scheduling of vertices and the improved parallelism resulting from the BSP model.

6.4.3 Topic-sensitive PageRank

To demonstrate the benefit of the simpler Eager execution policy, we implemented the topic-sensitive PageRank (TPR) algorithm [14] in GRACE. We also implemented this algorithm in GraphLab [21] with both the FIFO scheduler and the priority scheduler for comparison. Figure 14 shows the convergence results of both GRACE and GraphLab. We plot on the y-axis the average L1 distance between the converged graph and the snapshot with respect to the number of vertex updates so far. All asynchronous implementations converge faster than the synchronous implementation, and the high-overhead priority of GraphLab and the Prior execution of GRACE converge faster than the low-overhead FIFO scheduler of GraphLab or the Eager policy of GRACE. In addition, GRACE and GraphLab converge at similar rates with either low-overhead or high-overhead scheduling methods.

Figures 12 and 13 show the running time and the corresponding speedup of these asynchronous implementations on up to 16 worker threads. The Eager policy outperforms the Prior policy although it has more update function calls. This is because Topic PR is computationally light: although the Prior policy results in fewer vertex updates, this benefit is outweighed by its high scheduling overhead. We omit the result of the priority scheduler in GraphLab since it does not benefit from multiple processors: on 32 worker threads the running time (654.7 seconds) decreases by only less than 7 percent compared to the running time of 1 thread (703.3 seconds).

6.4.4 Social Community Detection

To illustrate the effectiveness of the GaussSeidel execution policy, we use GRACE to implement a social community detection (SCD) algorithm for large-scale networks [19, 29].

Figure 17 shows the convergence of the algorithm using the Jacobi and GaussSeidel execution policies. On the y-axis we plot the ratio of incorrect labeled vertices at the end of each iteration compared with the snapshot upon convergence. As we can see, using the Gauss-Seidel method we converge to a fixpoint after only 5 iterations; otherwise convergence is much slower, requiring more than 100 iterations to reach a fixpoint.

Figure 15 and 16 illustrate the running time and the multicore speedup compared with the sweep scheduler from GraphLab, which is a low-overhead static scheduling method. We observe that by following the BSP model for execution, GRACE achieves slightly better performance.

6.5 Results for Speedup

As discussed in Section 6.2.2, for light applications, which do little computation on each vertex, we expect limited scaling above a certain number of worker threads due to the memory bandwidth bottleneck. We have already observed this bottleneck for the SfM, topic-sensitive PageRank, and social community detection applications in Section 6.4, and we now investigate this issue further.

We start with image restoration, an application with a high ratio of computation to data access. Figure 18 shows the speedup for this application up to 32 worker threads. We observe that both the Eager and the Prior execution policies have nearly linear speedup. In GraphLab, FIFO and splash scheduling achieve similar speedup, but GraphLab's priority scheduler hits a wall around 7 threads.

Our remaining three applications have a low ratio of computation to data access. We show results here from topic-sensitive PageRank as one representative application; results from the other two graph applications are similar. As shown in Figure 19, since the computation for each fetched byte from the memory is very light, the speedup slows down after 12 threads for both GraphLab and GRACE as our execution becomes memory-bound: we cannot supply enough data to the additional processors to achieve further speedup. We illustrate this phenomenon by adding some (unnecessary) extra computation to the update function of TPR to create

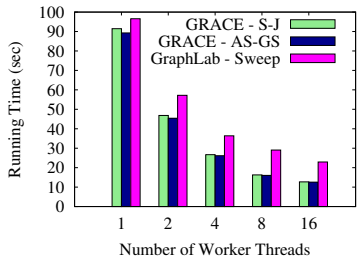

Figure 15: LP Running Time

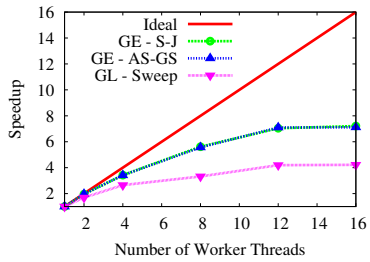

Figure 16: LP Speedup

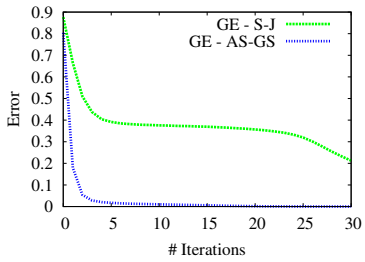

Figure 17: LP Convergence

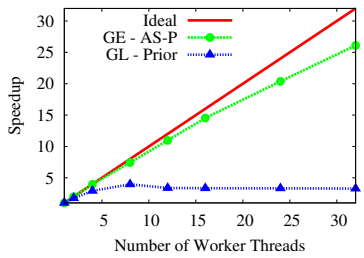

Figure 18: IR Speedup$_{32core}$

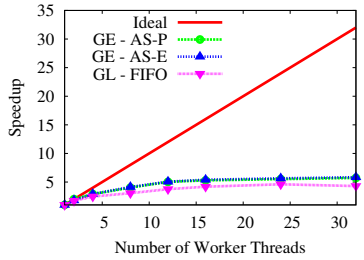

Figure 19: TPR Speedup$_{32core}$

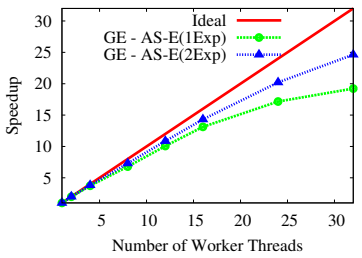

Figure 20: TPR$_{loaded}$ Speedup$_{32core}$

a special version that we call "Loaded Topic PR." The speedup of Loaded Topic PR, shown in Figure 20 is close to linear up to 32 threads. Since both Eager and Prior have similar speedup, we only show the speedup for Eager in the figure. In Eager(1Exp) we have added one extra `exp` function call for each plus operation when updating the preference vector, and in Eager(2Exp) we added two extra `exp` function calls for each plus operation. We can observe that although Loaded Topic PR with two `exp` function calls is about 6.75 times slower than the original Topic PR on a single core, it only takes 1.6 times longer on 32 processors, further demonstrating that the memory bottleneck is inhibiting speedup.

7. RELATED WORK

Much of the existing work on iterative graph programming systems is based on the Bulk Synchronous Parallel (BSP) model which uses global synchronization between iterations. Twister [11], PEGASUS [15], Pregel [23] and PrIter[38] build on MapReduce [10], which is inspired by the BSP model. Naiad [24] is an iterative incremental version of DryadLINQ [35] with one additional fixpoint operator, in which the stop criterion is checked at the end of each tick. HaLoop [8] extends the Hadoop library [1] by adding programming support for iterative computations and making the task scheduler loop-aware. Piccolo [28] and Spark [37] also follow an iterative computation pattern although with a higher-level programming interface than MapReduce/DryadLINQ; they also keep data in memory instead of writing it to disk. All these frameworks provide only synchronous iterative execution for graph processing (e.g., update all vertices in each tick) — except PrIter, which allows selective execution. However, since PrIter is based on MapReduce, it is forced to use an expensive shuffle operation at the end of each tick, and it requires the application to be written in an incremental form, which may not be suitable for all applications (such as a Belief Propagation). In addition, because of the underlying MapReduce framework, PrIter cannot support Gauss-Seidel update methods.

GraphLab [21, 22] was the first approach to use a general asynchronous model for execution. In their execution model, computation is organized into tasks, with each task corresponding to the update function of a scope of a vertex. The scope of a vertex includes itself, its edges and its adjacent vertices. Different scheduling algorithms are provided for ordering updates from tasks. GraphLab forces programmers to think and code in an asynchronous way, having to consider the low level issues of parallelism such as memory consistency and concurrency. For example, users have to choose one from the pre-defined consistency models for executing their applications.

Galois [17], which is targeted at irregular computation in general, is based on optimistic set iterations. Users are required to provide commutativity semantics and undo operations between different function calls in order to eliminate data races when multiple iterators access the same data item simultaneously. As a result, Galois also requires users to take care of low level concurrency issues such as data sharing and deadlock avoidance.

Another category of parallel graph processing systems are graph querying systems, also called graph databases. Such systems include HyperGraph DB [2], InfiniteGraph DB [3], Neo4j [4] and Horton [30], just to name a few. Their primary goal is to provide a high level interface for processing queries against a graph. Thus, many database techniques such as indexing and query optimizations can be applied. In order to implement a graph processing application over these systems, users can only interactively call some system provided functions such as BFS/DFS traversals or Dijkstra's algorithms. This restricts programmability and applicability of these systems for iterative graph processing applications.

8. CONCLUSIONS

In this paper we have presented GRACE, a new parallel graph processing framework. GRACE provides synchronous programming with asynchronous execution while achieving near-linear parallel speedup for applications that are not memory-bound. In particular, GRACE can achieve a convergence rate that is comparable to asynchronous execution while having a comparable or better multi-core speedup compared to state-of-the-art asynchronous engines and manually written code.

In future work, we would like to scale GRACE beyond a single machine, and we would like to investigate how we can further

improve performance on today's NUMA architectures.

Acknowledgements. We thank David Bindel for helpful discussions, and we thank the anonymous CIDR reviewers for their insightful comments which helped us to improve this paper. This research has been supported by the NSF under Grants CNS-1012593, IIS-0911036, by a Google Research Award, by the AFOSR under Award FA9550-10-1-0202 and by the iAd Project funded by the Research Council of Norway. Any opinions, findings, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the sponsors.

9. REFERENCES

[1] Hadoop. http://hadoop.apache.org/core/.

[2] Hypergraphdb. http://www.hypergraphdb.org/.

[3] Infinitegraph: The distributed graph database. http://www.infinitegraph.com/.

[4] Neo4j: Nosql for the enterprise. http://neo4j.org/.

[5] USC-SIPI image database. sipi.usc.edu/database/.

[6] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and distributed computation: numerical methods*. Prentice-Hall, Inc., 1989.

[7] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

[8] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1):285–296, 2010.

[9] D. Crandall, A. Owens, N. Snavely, and D. P. Huttenlocher. Discrete-Continuous optimization for large-scale structure from motion. In *CVPR*, pages 3001–3008, 2011.

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

[11] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: A runtime for iterative mapreduce. In *HPDC*, pages 810–818, 2010.

[12] G. Elidan, I. McGraw, and D. Koller. Residual belief propagation: Informed scheduling for asynchronous message passing. In *UAI*, 2006.

[13] J. Gonzalez, Y. Low, and C. Guestrin. Residual splash for optimally parallelizing belief propagation. In *AISTATS*, pages 177–184, 2009.

[14] T. H. Haveliwala. Topic-sensitive pagerank. In *WWW*, pages 517–526, 2002.

[15] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system. In *ICDM*, pages 229–238, 2009.

[16] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., 2005.

[17] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, pages 211–222, 2007.

[18] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1), 2009.

[19] I. X. Y. Leung, P. Hui, P. Liò, and J. Crowcroft. Towards real-time community detection in large networks. *Physical Review E*, 79:066107, 2009.

[20] F. Lin and W. W. Cohen. Semi-supervised classification of network data using very few labels. In *ASONAM*, pages 192–199, 2010.

[21] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. In *UAI*, pages 340–349, 2010.

[22] 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.

[23] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[24] F. McSherry, D. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *CIDR*, 2013.

[25] K. P. Murphy, Y. Weiss, and M. I. Jordan. Loopy belief propagation for approximate inference: An empirical study. In *UAI*, pages 467–475, 1999.

[26] J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann Publishers Inc., 1988.

[27] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *PLDI*, pages 12–25, 2011.

[28] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, pages 293–306, 2010.

[29] U. N. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76:036106.

[30] M. Sarwat, S. Elnikety, Y. He, and G. Kliot. Horton: Online query execution engine for large distributed graphs. In *ICDE*, pages 1289–1292, 2012.

[31] R. Szeliski, R. Zabih, D. Scharstein, O. Veksler, V. Kolmogorov, A. Agarwala, M. Tappen, and C. Rother. A comparative study of energy minimization methods for markov random fields with smoothness-based priors. *IEEE Trans. Pattern Anal. Mach. Intell.*, 30, 2008.

[32] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[33] L. G. Valiant. A bridging model for multi-core computing. *J. Comput. Syst. Sci.*, 77(1):154–166, 2011.

[34] G. Wang, M. A. V. Salles, B. Sowell, X. Wang, T. Cao, A. J. Demers, J. Gehrke, and W. M. White. Behavioral simulations in mapreduce. *PVLDB*, 3(1):952–963, 2010.

[35] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Úlfar Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, pages 1–14, 2008.

[36] A. Yzelman and R. H. Bisseling. An object-oriented bulk synchronous parallel library for multicore programming. *Concurrency and Computation: Practice and Experience*, 24(5):533–553, 2012.

[37] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.

[38] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Priter: A distributed framework for prioritized iterative computations. In *SOCC*, 2011.