To be published at the International Semantic Web Conference (ISWC) 2010.
The framework will be available for download on November 5, 2010.

Signal/Collect: Graph Algorithms for the (Semantic) Web

Philip Stutz[1], Abraham Bernstein[1], and William Cohen[2]

[1] DDIS, Department of Informatics, University of Zurich, Zurich, Switzerland
[2] Machine Learning Department, Carnegie Mellon University, Pittsburgh, PA
{stutz,bernstein}@ifi.uzh.ch
wcohen@cs.cmu.edu

Abstract. The Semantic Web graph is growing at an incredible pace, enabling opportunities to discover new knowledge by interlinking and analyzing previously unconnected data sets. This confronts researchers with a conundrum: Whilst the data is available the programming models that facilitate scalability and the infrastructure to run various algorithms on the graph are missing.
Some use MapReduce – a good solution for many problems. However, even some simple iterative graph algorithms do not map nicely to that programming model requiring programmers to shoehorn their problem to the MapReduce model.
This paper presents the Signal/Collect programming model for synchronous and asynchronous graph algorithms. We demonstrate that this abstraction can capture the essence of many algorithms on graphs in a concise and elegant way by giving Signal/Collect adaptations of various relevant algorithms. Furthermore, we built and evaluated a prototype Signal/Collect framework that executes algorithms in our programming model. We empirically show that this prototype transparently scales and that guiding computations by scoring as well as asynchronicity can greatly improve the convergence of some example algorithms. We released the framework under the Apache License 2.0 (at http://www.ifi.uzh.ch/ddis/research/sc).

1 Introduction

The Semantic Web confronts researchers and practitioners with increasing data set sizes. One approach to deal with this problem is to hope for the computational capabilities of computers to grow faster than the datasets relying on Moore's Law [1]. This approach is somewhat impractical as it makes current work rather tedious and relies on the hope that Moore's Law will be sustainable and will outpace the growth of data—both of which are unsure prospects. As a consequence, many researchers have tried to use parallelism to improve the performance of Semantic Web computational tasks. Hereby, they used two avenues of investigation. On the one hand, they have tried to use distributed computing programming models such as MapReduce [2] to achieve their goals [3, 4]. This,

usually, requires to cumbersomely shoehorn their computation to the programming model. In the case of MapReduce the typed graphs of the Semantic Web need to be inconveniently mapped to the key/value-pair programming model. On the other hand, many have used low-level distributed computing primitives such as message passing interfaces [5], clusters [6], or distributed hash trees [7], which requires the building of the whole infrastructure for Semantic Web graph processing from scratch—a tedious task.

This paper proposes a *novel and scalable programming model for typed graphs*. The core idea lies in the realization, that most computations on Semantic Web data involve the passage of (1) some kind of information between the resources (or vertices) along the properties (or edges) of the RDF graph and (2) some computation at the vertices of the RDF graph. Specifically, we propose a programming model where vertices send *signals* along the property-defined edges of a compute graph and then a *collect* function gathers the incoming signals at the vertices to perform some computation. Given the two core elements we call our model SIGNAL/COLLECT.

This programming model allows an elegant and concise definition of programming tasks on typed graphs that a suitable execution framework can process transparently in a distributed fashion. In some cases the framework can exploit asynchronous execution to further speed up the accomplishment of the task. As such SIGNAL/COLLECT provides a natural programming model for the Semantic Web, which can serve as an alternative to paradigms such as MapReduce.

Given the above, the contributions of this paper are the following: First, we introduce an *elegant and concise programming model for Semantic Web computing tasks*. We show the elegance and conciseness by presenting the model and providing a number of typical algorithm examples. Second, we empirically show that a simple *execution framework is able to transparently parallelize* SIGNAL/-COLLECT *computations* and can be *simply initialized with SPARQL queries*. Third, we empirically show that the exploitation of *asynchronous execution can further increase the performance and convergence* of an already parallel algorithm.

The remainder of the paper is organized as follows: Section 2 formally introduces the programming model and its extensions. Section 3 describes a number of increasingly complex graph algorithms to illustrate the elegance and conciseness of the programming model. We then introduce an actual implementation and evaluate the scalability, the impact of guiding computations by scoring and asynchronous computations in Sections 4 and 5. We close with a discussion of the related and future work.

2 The Signal/Collect Programming Model

The general intuition behind our SIGNAL/COLLECT programming model is that all computations are executed on a compute graph, where the vertices are the computational units that interact by the means of signals that flow along the edges. All computations in the vertices are accomplished by collecting the in-

coming signals and performing some computation on them employing, possibly, some vertex-state, and then signaling their neighbors in the compute graph.

To give a more concrete example: imagine a graph with RDFS classes as vertices and edges from superclasses to subclasses (i.e., `rdfs:subClassOf` triples). Every vertex has a set of superclasses as state, which initially only contains itself. Now all the superclasses send their own states as signals to their subclasses, which collect those signals by setting their own new state to the union of the old state and all signals received. It is easy to imagine how these steps, when repeatedly executed, iteratively compute the transitive closure of the `rdfs:subClassOf` relationship in the vertex states.

SIGNAL/COLLECT provides an elegant and concise abstraction for describing such graph-based algorithms. So far, however, we glossed over a number of important details which we elaborate in this section. We will introduce a formal definition of the basic structures of the SIGNAL/COLLECT programming model and continue by specifying the synchronous/asynchronous execution modes for computations as well as extending the basic model to support them.

2.1 A Formal Definition of the Signal/Collect Structures

The basis for any SIGNAL/COLLECT computation is the *compute graph*

$$G = (V, E),$$

where V is the set of vertices and E the set of edges in G. Every *vertex* $\mathtt{v} \in V$ has the following attributes:

v.id a unique id.
v.state the current vertex state which represents computational intermediate results.
v.outgoingEdges a list of all edges $\mathtt{e} \in E$ with $\mathtt{e.source} = \mathtt{v}$.
v.signalMap a map with the ids of vertices as keys and signals as values. Every key represents the id of a neighboring vertex and its value represents the most recently received signal from that neighbor. We will use the alias `v.signals` to refer to the list of values in `v.signalMap`.
v.uncollectedSignals a list of signals that arrived since the collect operation was last executed on this vertex.

Every *edge* $\mathtt{e} \in E$ has the following attributes:

e.source the source vertex
e.sourceId id of the source vertex
e.targetId id of the target vertex

The default vertex type also defines an abstract **collect** function and the default edge type defines an abstract **signal** function. To specify an algorithm in the SIGNAL/COLLECT programming model the default types have to be extended with implementations of those functions. The **collect** function calculates a new vertex state, while the **signal** function calculates the signal that will be sent along an edge.

We have now defined the basic structures of the programming model. In order to completely define a SIGNAL/COLLECT computation we still need to describe how to execute computations on them.

2.2 The Computation Model and Extensions

In this section we will specify how both synchronous and asynchronous computations are executed in the SIGNAL/COLLECT programming model. Also we will provide extensions to the core model.

We will use the attribute `target` on edges to denote the target vertex, but this is only to specify the behavior without having to describe how signals are relayed.
We first define an additional attribute `lastSignalState` and two additional functions on all vertices $\mathtt{v} \in V$, which will enable us to describe computations in SIGNAL/COLLECT:

v.executeSignalOperation
    `lastSignalState` := `state`
    for all (`e` ∈ `outgoingEdges`) **do**
        `e.target.uncollectedSignals.append(e.signal)`
        `e.target.signalMap.put(e.sourceId, e.signal)`
    end for

v.executeCollectOperation
    `state` := `collect`
    `uncollectedSignals` := Nil

With these functions we are now able to describe a synchronous SIGNAL/-COLLECT execution.

Synchronous Execution A synchronous computation is specified in Algorithm 1. Its parameter num_iterations defines the number of iterations (computation steps the algorithm is going to perform. Everything inside the inner loops can

Algorithm 1 Synchronous execution of SIGNAL/COLLECT

```
for i ← 1..num_iterations do
    for all v ∈ V parallel do
        v.executeSignalOperation
    end for
    for all v ∈ V parallel do
        v.executeCollectOperation
    end for
end for
```

be executed in parallel, with a global synchronization between the signaling and collecting phases, which is similar to computations in Pregel [8]. This parallel programming model is more generally referred to as Bulk Synchronous Parallel (BSP).

This specification allows the efficient execution of algorithms, where every vertex is equally involved in all steps of the computation. However, in many algorithms only a subset of the vertices is involved in each part of the computation. In order to be able to define a computational model that enables us to

guide the computation and give priority to more "important" operations, we will introduce scoring.

Extension 1: Score-Guided Execution In order to enable the scoring (or prioritizing) of signal/collect operations, we need to extend the core structures of the SIGNAL/COLLECT programming model. This is why we define two additional functions on all vertices $\mathtt{v} \in V$:

v.scoreSignal : **Double**
: is a function that calculates a number that reflects how important it is for this vertex to signal. The result of this function is only allowed to change when the `v.state` changes. Its default implementation returns 0 if `state` = `lastSignalState` and 1 otherwise. This captures the intuition that it is desirable to inform the neighbors iff the state has changed since they were informed last.

v.scoreCollect : **Double**
: is a function that calculates a number that reflects how important it is for this vertex to collect. The result of this function is only allowed to change when `uncollectedSignals` changes. Its default implementation returns `uncollectedSignals.size`. This captures the intuition that the more new information is available, the more important it is to update the state.

The default implementations can be overridden with functions that capture the algorithm-specific notion of "importance" more accurately.

Now that we have extended the basic model with scoring we specify a score-guided synchronous execution of a SIGNAL/COLLECT computation in Algorithm 2. There are three parameters that influence when the algorithm stops:

Algorithm 2 Score-guided synchronous execution of SIGNAL/COLLECT

```
done := false
while iterations < num_iterations and !done do
    done := true
    iterations := iterations +1
    for all v ∈ V parallel do
        if (v.signalScore > signal_threshold) then
            done := false
            v.executeSignalOperation
        end if
    end for
    for all v ∈ V parallel do
        if (v.collectScore > collect_threshold) then
            done := false
            v.executeCollectOperation
        end if
    end for
end while
```

`signal_threshold` and `collect_threshold` which set a minimum level of "importance" for operations that get executed and `num_iterations`, which limits the number of computation steps. This means that the algorithm is guaranteed to stop, either because the maximum number of iterations is reached or because there are no operations anymore that score higher than the threshold. If the second condition is fulfilled, we say that the algorithm has converged.

Asynchronous Execution We can now also define an asynchronous execution which gives no guarantees about the order of execution or the ratio of signal/-collect operations. We referred to the first two execution modes as synchronous because they guarantee that all vertices are in the same "loop" at the same time. In a synchronous execution it can never happen that one vertex executes a signal operation while another vertex is executing a collect operation, because the switch from one phase to the other is globally synchronized. In an asynchronous computation, in contrast, no such guarantees exist.

Algorithm 3 Asynchronous execution of SIGNAL/COLLECT

```
ops := 0
while [ops < num_ops] ∧ [∃v ∈ V((v.signalScore > signal_threshold) ∨
(v.collectScore > collect_threshold))] do
    S := radomly choose subset of V
    for all v ∈S parallel do
        Randomly call either
            v.executeSignalOperation or v.executeCollectOperation
            assuming respective threshold is reached
        ops := ops + 1
    end for
end while
```

As shown in Algorithm 3 there are again three parameters that influence when the asynchronous algorithm stops: `signal_threshold` and `collect_threshold`, which have the same function as in the synchronous case and `num_ops` which instead of the number of iterations limits the number of operations executed. Again this guarantees that an asynchronous execution terminates, either because the maximum number of operations is exceeded or because it converged. The purpose of Algorithm 3 is not to be executed directly, but to specify what kind of restrictions are guaranteed (by an execution environment) during asynchronous execution. This freedom is useful, because if an algorithm no longer has to maintain the execution order of operations then one is able use different scheduling strategies for those operations.

Extension 2: Scheduled Asynchronous Operations As an extension to the asynchronous execution we can define operation schedulers that optimize certain measures. For example we can define an eager scheduler (see Algorithm 4 in Section 4) that will execute the signal operation of a vertex immediately after

the collect operation of that same vertex. This allows other vertices to receive those signals sooner. Another example is a scheduler that gives priority to signals that have high scores by only executing signals with at least an average score. Depending on the algorithm this can result in fewer operations being executed, which, depending on the operation costs, impact of scheduling on convergence and the cost of scheduling itself, can pay off.

Extension 3: Multiple Vertex/Edge Types Some algorithms, for example operating on OWL ontologies in RDF or bipartite factor graphs, require several kinds of vertices and edges with different associated functions for signaling, collecting, etc. This is why a compute graph can contain vertices and edges that have different types.

Extension 4: Result Processing Results are processed by a `resultProcessing` function defined on the default vertex type. The default implementation does nothing and is meant to be overridden. This function gets executed on all vertices once the computation has ended.

Extension 5: Weights The model supports weights on edges and the vertices keep track of the sum of weights of outgoing edges. It is also possible to extend the default edge/vertex type with labels or whatever additional attributes or functions should be required.

Extension 6: Conditional Edges & Computation Phases Edges can be extended to only send a signal when certain conditions have been met. A possible condition is that a source vertex has received a convergence signal from an aggregation vertex, which can be used to trigger a next computation phase in the target vertices. Another use for this feature is to avoid sending the same signal repeatedly.

Feature: Aggregation Sometimes it is desirable to aggregate over the state of multiple vertices to, for example, obtain a global convergence criterium. This can be easily achieved by introducing an aggregation vertex that receives a signal from all the vertices it needs to aggregate.

We have now specified the structures and execution model of the SIGNAL/-COLLECT programming model. In the next section we show the usefulness of this programming model by giving implementations for various algorithms.

3 Algorithms in Signal/Collect

We argue that the SIGNAL/COLLECT programming model is a useful abstraction. There are many important algorithms that can be expressed in a concise and elegant way, which proves that this abstraction captures the essence of many computations on graphs indeed.

We demonstrate these characteristics of the SIGNAL/COLLECT programming model by giving examples (written in Scala-like[3] pseudocode, where the initialization of class variables is passed in parentheses) of several interesting algorithms expressed in SIGNAL/COLLECT. Note that not all algorithms work with all execution modes. Vertex coloring for example does not converge without score-guidance.

Single-source shortest path. Here the vertex states represent the shortest currently known path from the path-source, edge weights are used to represent distance. The signals represent the total path length of the shortest currently known path from the path-source to `e.target` that passes through `e`. In the Semantic Web context this algorithm can be used to compute the Rada shortest path distance along subclass vertices, which is sometimes used to denote the similarity between two classes.

```
class Location(id: Any, initalState: Int) extends Vertex {
  def collect: Integer = min(state, min(uncollectedSignals))
}
class Path(sourceId: Any, targetId: Any) extends Edge {
  def signal: Integer = source.state+weight
}
```

RDFS subclass inference. A vertex represents an RDFS class. The vertex state represents the set of currently known superclasses of the given vertex. As the edges just signal the set of currently known superclasses of the class represented by the source vertex, one can simply use the predefined StateForwarder edge, which has the signal function: `def signal = source.state`. The compute graph can be built with edges from vertices representing super-classes to vertices representing sub-classes, which can easily be done with a SPARQL query (see full PageRank example code in Figure 1, Section 4).

```
class RdfsClass(id: String, initialState=Set(iri)) extends Vertex {
  def collect: Set[String] = state ∪ ⋃_{s∈uncollectedSignals} s
}
```

Vertex coloring. The following algorithm solves the vertex coloring problem by assigning to each vertex a random color. The vertices keep switching to different random colors wherever conflicts with neighbors remain. The predefined StateForwarder edges are used again to signal the color of a vertex to its neighbors.

```
class Colored(id: Any, initialState=randomColor) extends Vertex {
```

[3] http://www.scala-lang.org/

```
  def collect: Int = {
    if (signals.contains(state)) randomColorExcept(state) else state
  }
}
```

PageRank [9]. The vertex state represents the current pagerank of a vertex. The signals represent the rank transferred from `e.source` to `e.target`.

```
class Document(id: Any, initialState=0.15) extends Vertex {
  def collect: Double = 0.15+0.85* ∑_{s∈v.signalMap.values} s
}
class Citation(sourceId: Any, targetId: Any) extends Edge {
  def signal: Double = weight*source.state / source.sumOfOutWeights
}
```

Loopy Belief Propagation [10]. Loopy belief propagation subsumes inferencing on Relational Probabilistic Models. These can be used to combine logical and probabilistic inference on Semantic Web data—a highly desirable goal.

Because of space constraints we just convey the intuition: A factor graph can be defined in SIGNAL/COLLECT with two vertex types Factor and Variable and edge types FactorToVariable and VariableToFactor. Loopy belief propagation is a message passing algorithm on a factor graph where messages are passed back and forth between factor and variable vertices. Those messages in turn are calculated from the messages received by the respective factor/variable. In the simplest adaptation we put the code that computes those messages into the signal functions of the edges. These functions can directly calculate the new outgoing signals from the received signals in the signalMap of the source vertex.

All current evidence also indicates (but we have not tried it yet) that SIGNAL/COLLECT can straightforwardly implement the general sum-product (GSP) algorithm [11]. According to Kschischang et al. [11] a wide variety of algorithms, such as the forward/backward algorithm, the Viterbi algorithm, the iterative turbo decoding algorithm, Pearls belief propagation algorithm for Bayesian networks, the Kalman filter, and certain fast Fourier transform (FFT) algorithms can be derived as specific instances of the GSP algorithm.

In this subsection we demonstrated that the SIGNAL/COLLECT programming model is a useful abstraction by giving examples of several interesting algorithms, which we were able to express in a concise and elegant way. Note that whilst not all of them are initially recognizable as typical Semantic Web approaches they all provide important functionality.

In the next section we are going to evaluate the properties of a prototype of the SIGNAL/COLLECT framework which can execute algorithms such as the ones we just described.

4 The Signal/Collect Framework — An Implementation

The SIGNAL/COLLECT framework provides an execution platform for algorithms specified according to the SIGNAL/COLLECT programming model. This is analogous to the Hadoop MapReduce framework which executes algorithms expressed in the MapReduce programming model. The framework has been implemented in Scala—a fusion of the object-oriented and functional programming paradigms running on the Java Virtual Machine. We released the framework under the Apache License 2.0 (http://www.ifi.uzh.ch/ddis/research/sc).

Parallel Computations: The current implementation of the framework can parallelize computations exploiting multiple processor cores of one computer and shared memory for efficient signal passing. To that end we assign the vertices to worker threads that are each responsible for a part of the graph. We use a hash function on the vertex ids for the mapping of vertices to workers, edges are always assigned to the same worker as their source vertex.

We implemented the synchronous computation similar to [8] with a master that orchestrates the synchronized execution of signal/collect steps for all worker threads.

Asynchronous Scheduling: In an asynchronous computation every worker decides on its own which operations to execute. For this purpose every worker has a scheduler that determines the order in which the signal/collect operations get executed. We experimented with different schedulers for the signal/collect operations in the asynchronous case. Every computation in asynchronous mode starts with one synchronous score-guided signal/collect step, as this improved performance for the algorithms we analyzed. After that a scheduler takes over. The "eager" scheduler (Algorithm 4) for example tries to have a vertex signal as soon as possible after collecting.

Algorithm 4 "Eager" scheduler: tries to signal right after collection

```
for all v ∈V do
    if (v.collectScore > collect_threshold) then
        v.executeCollectOperation
        if (v.signalScore > signal_threshold) then
            v.executeSignalOperation
        end if
    end if
end for
```

We also experimented with other schedulers that, for example, only execute signal operations that score above or equal to the average signal score ("above average" scheduler).

Specifying Graphs: The PageRank example. In order to specify compute graphs one needs to define the necessary elements of the SIGNAL/COLLECT programming model. Consider the SIGNAL/COLLECT implementation of the PageRank algorithm optimized with residual scoring on SwetoDblp[4] cita-

[4] http://knoesis.wright.edu/library/ontologies/swetodblp

tions shown in Figure 1. First, the Figure specifies the PageRank algorithm by defining both the `collect` and `signal` functions. Second, the `Algorithm` object initializes a score-guided and synchronous compute-graph by iterating over the answers of a SPARQL query and then executes the algorithm with a signal threshold of 0 using `computeGraph.execute(.)`. Note that this is the executable code and not simplified pseudocode.

Algorithm

```
class Document(id: Any) extends Vertex(id, 0.15) {
  def collect = 0.15 + 0.85 * signals[Double].foldLeft(0.0)(_ + _)
  override def processResult = if (state > 5) println(id + ": " + state)
  override def scoreSignal = (state - lastSignalState.getOrElse(0)).abs
}
```

```
class Citation(citer: Any, cited: Any) extends Edge(citer, cited) {
  override type SourceVertexType = Document
  def signal = source.state * weight / source.sumOfOutWeights
}
```

Initialization

```
object Algorithm {
  def executeCitationRank(db: SparqlAccessor) {
    val computeGraph = new ComputeGraph(ScoreGuidedSynchronous)
    val citations = new SparqlTuples(db, "select ?source ?target where {"
      + "?source <http://lsdis.cs.uga.edu/projects/semdis/opus#cites> ?target}")
```

Execution

```
    citations foreach {
      case (citer, cited) =>
        computeGraph.addVertex[Document](citer)
        computeGraph.addVertex[Document](cited)
        computeGraph.addEdge[Citation](citer, cited)
    }
    computeGraph.execute(signalThreshold = 0)
  }
}
```

Fig. 1. Complete implementation of the PageRank algorithm on citations, including residual scoring and result processing. Written in Scala and executable as-is on the framework.

5 Evaluation

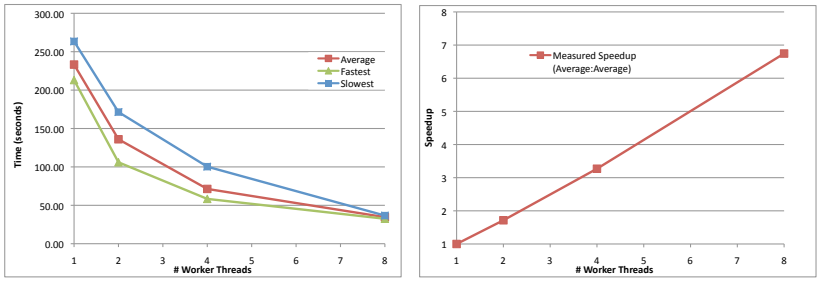
Having established the elegance and conciseness of the SIGNAL/COLLECT programming model by example in Sections 3/4 we can now turn to validate the second and third of our claims: the scalability/transparency of our parallelization framework and the ability of our programming model to exploit score-guidance and asynchronous execution to further improve performance.

5.1 Scalability

To establish the ability of the SIGNAL/COLLECT framework to transparently scale we evaluated its performance when running the single-source shortest path algorithm ("above average" score-guided asynchronously) with a varying number of worker threads on a computer with two quad-core Intel® Xeon® X5570 processors (turbo boost & hyper-threading disabled to reduce confounding effects) and 72 GB RAM.

Figure 2(a) shows the average, fastest and slowest running-times over 10 executions, while Figure 2(b) shows that the performance scaled almost linearly with the number of worker threads used. Bearing the limitations discussed in Section 5.4 below we have, hence, established that *given the right algorithm and graph our programming model and framework can provide excellent scalability.*

Fig. 2. Scalability of Signal/Collect: Single-source shortest path on a randomly generated graph with a log-normal distribution of out-degrees. The graph had 1 million vertices, 94 million edges and a longest path of 5. Results of 10 executions for each number of worker threads.



(a) Scaling with additional workers (b) Parallel speedup

5.2 Score-Guided Computations

In order to evaluate the impact of guiding computations by scoring and, hence, establish its usefulness we ran PageRank (as shown in Figure 1) with and without score-guiding (residual scoring, signal_threshold=0.001) on two different graphs. The hardware we used for all further evaluations was a MacBook Pro i7 2.66 GHz (2 cores, hyper-threading enabled) with 8 GB RAM running four worker threads. Also we used a newer version of the framework than in the previous experiment. Figures 3 and 4 show the averages over 10 executions for each algorithm and the error bars indicate min/max values.

These results show that score-guided execution in general performed very well on the less densely connected citation graph, where some parts of the graph probably converged faster than others. On the densely connected generated graph the synchronous version performed comparably to the score-guided algorithms. We can conclude that given a suitable combination of algorithm and graph, *score-guidance can improve convergence significantly by focusing the computation only on the parts of the graph that still require it.*

5.3 Asynchronous vs. Synchronous

The results in Figure 4 suggest that the performance of the asynchronous version is highly dependent on the scheduling. For this combination of algorithm and graph the "eager" asynchronous version performed well and outperformed the synchronous approach, while the "above average" asynchronous scheduler performed poorly. Hence, more evaluations are required to determine which combinations of scheduling algorithms and graph algorithms/structures work well.

Fig. 3. PageRank on SwetoDblp citations, 22 387 vertices (publications) connected by 112 303 edges (citations)

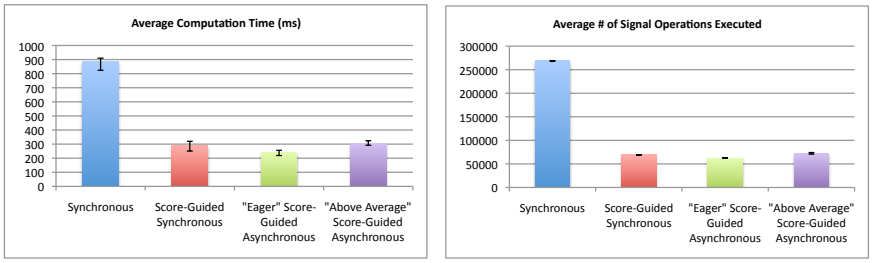
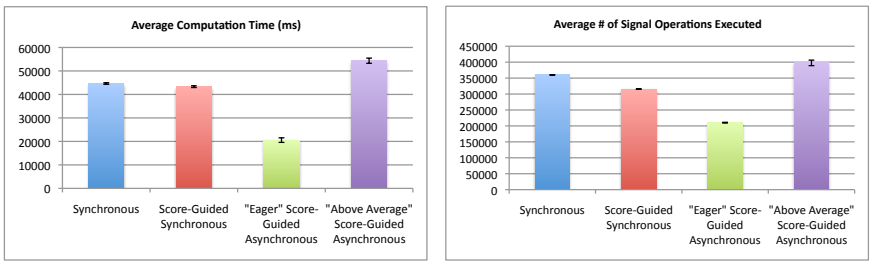


Fig. 4. PageRank on a generated graph with log-normal distributed out-degrees (drawn from $e^{\mu+\sigma N}$, where $\mu = 4$, $\sigma = 1.3$ and N is drawn from a standard normal distribution). The generated graph has 100 000 vertices connected by 1 284 495 edges.



To establish that for some algorithms the asynchronous version outperforms the synchronous we ran the vertex coloring algorithm introduced in Section 3 on a generated graph. The results in Table 1 show that the asynchronous version converges quickly for some problems, where the synchronous version fails to converge (within a reasonable amount of time). Other algorithms share this property: Koller and Friedman note that some asynchronous loopy belief propagation computations converge where the synchronous computations keep oscillating. They summarize [12, p. 408]: *"In practice an asynchronous message passing scheduling works significantly better than the synchronous approach. Moreover, even greater improvements can be obtained by scheduling messages in a guided way."*

Table 1. Vertex coloring on a generated graph with log-normal distributed out-degrees (drawn from $e^{\mu+\sigma N}$, where $\mu = 1$, $\sigma = 0.2$ and N is drawn from a standard normal distribution). The generated graph has 100 000 vertices connected by 554 118 edges. The table shows the average time (in milliseconds) over 10 executions it took to find a vertex coloring with the given number of colors. When the algorithm failed to converge in less than a minute (on average) the time was noted as "did not converge" (d.n.c).

Number of colors	5	6	7	8	9	10	11
"Eager" Score-Guided Asynchronous	d.n.c	12870	1690	1392	1243	1218	1046
Score-Guided Synchronous	d.n.c	d.n.c	d.n.c	d.n.c	d.n.c	2856	1876

5.4 Limitations—Threats to Validity

The main limitation of the evaluations above is that our current SIGNAL/COLLECT framework only runs on a single machine using shared memory for signaling. It is not entirely clear how the overhead of signaling across the network with the involved bandwidth and latency implications would impact the scalability of the prototype system. We do not expect this limitation to have an impact on the evaluations of score-guided and asynchronous execution.

In terms of scalability, we only ran our experiment on one large graph with an algorithm that has a very simple interaction pattern and many more vertices than worker threads and many edges per vertex. For a refined evaluation we need to investigate the impact of different graph structures and interaction scenarios.

Note also, that we only ran our second experiment on one algorithm. Before analyzing the impact of all important factors (algorithm, graph, scoring functions/thresholds, number of worker threads, asynchronous operation scheduling, etc.) it is difficult to make a general statement about the trade-offs involved with regard to guided vs. unguided and synchronous vs. asynchronous computations.

6 Related Work

Many general programming models for distributed computing have been presented. Most notable is the MapReduce [2] programming model, which is based on parallel operations on sets of key-value pairs. The Hadoop MapReduce framework[5] has been used by [3, 4] for scalable RDFS/OWL reasoning. The big disadvantage of the MapReduce model is that it is based on key-value pairs requiring a translation of Semantic Web tasks to this abstraction. Also, the programming model was not designed with iterated executions in mind and if it is used iteratively, the model is limited to synchronous execution.

Most closely related to our programming model is Pregel [8]: a system developed by Google for large-scale graph processing. It has been shown to scale to graphs with billions of vertices/edges via distribution to thousands of commodity PCs. Its limitations are that it only handles synchronous computations, can only support graphs with one kind of vertex sharing a single "compute" function, and edges are not first class citizens. As we have seen in our evaluation, score-guided asynchronous computations are essential for some graph computations. Pregel's limitation to one vertex type makes the implementation of algorithms employing multiple kinds of vertices rather tedious.

The concurrency model of the asynchronous SIGNAL/COLLECT computation was inspired by the actor formalism [13], in which many processor objects take part in a computation and can only influence each other via messaging. This bears a lot of similarity to vertices in SIGNAL/COLLECT, which do local computations and can only influence each other via signaling.

We did not find any other programming models specialized for parallel iterated computation on typed graphs (such as the Semantic Web). However, in

[5] http://hadoop.apache.org/mapreduce

addition to the use of generic distributed computing frameworks many have implemented their own distributed systems for Semantic Web tasks [6, 7]. Weaver and Hendler [5], e.g., present an RDFS closure using MPI—a low-level message passing interface. Oren et al. [14] have implemented a distributed reasoner using their own low-level primitives. We believe that most of these solutions could profit from our generic framework.

7 Limitations, Future Work and Conclusions

In order to master the onslaught of data the Semantic Web is in dire need of distributed computation paradigms. Current paradigms either have the problem that their programming model does not lend itself naturally to the typed graph based Semantic Web computation tasks or provide only low-level functionality requiring the tedious implementation of the whole functionality for every algorithm. This paper presented a novel, distributed, and scalable computing model for typed graphs called SIGNAL/COLLECT. We showed a framework that can be used to elegantly and concisely specify and execute a number of computations that are typical for the Semantic Web fully incorporating Semantic Web techniques such as SPARQL (to initialize the graph). We also showed that the programming model allows for scalable implementations given suitable algorithms and graphs. Lastly, we showed that the support for asynchronous execution of graph algorithms enables the convergence for some algorithms that will not converge in the synchronous case.

Whilst these results are remarkable SIGNAL/COLLECT is still at its beginning. First, we need to find the limitations of the programming model. Although it is suitable for computations on graphs it is, obviously, not quite as suitable for computations on lists. Second, we need to extend the framework for distribution and explore heuristics for the distribution of vertices in the compute graph to compute nodes. This is a non-trivial problem as signals transmitted across the network will incur significant latencies compared to signals transmitted in a shared memory setting. Consequently, the algorithms need to be robust against signal latency variance. Third, for the use outside research we need to build a framework that provides typical middle-ware services (such as distributed file-system access). We plan to investigate each of these areas in the future.

The Semantic Web is growing and so are the needs for processing its RDF-based data. Many have approached the call for processing these large-sized RDF graph data sets. Researchers have developed stores (or data bases) that scale to disk, have explored various means for computing the logical closure, and built large-scale systems. In order for large-scale processing of these data to go main-stream we need elegant programming models that allow for the concise formulation of a large amount of Semantic Web tasks. SIGNAL/COLLECT is such a programming model that, we believe, can serve the function as a general purpose Semantic Web infrastructure. As such, it has the potential to bring distributed computing transparently to the Semantic Web and become a major building block for future Semantic Web applications.

Acknowledgemement
We would like to thank Stefan Schurgast for using early prototypes of the framework and providing valuable feedback on its usage.

References

1. Moore, G.E.: Cramming more components onto integrated circuits. Electronics **38**(8) (1965)
2. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, Berkeley, CA, USA, USENIX Association (2004) 10–10
3. Urbani, J., Kotoulas, S., Oren, E., van Harmelen, F.: Scalable distributed reasoning using mapreduce. In: Proceedings of the ISWC '09. Volume 5823 of LNCS., Springer (2009)
4. Urbani, J., Kotoulas, S., Maassen, J., van Harmelen, F., Bal, H.E.: Owl reasoning with webpie: Calculating the closure of 100 billion triples. In Aroyo, L., Antoniou, G., Hyvönen, E., ten Teije, A., Stuckenschmidt, H., Cabral, L., Tudorache, T., eds.: ESWC (1). Volume 6088 of Lecture Notes in Computer Science., Springer (2010) 213–227
5. Weaver, J., Hendler, J.: Parallel materialization of the finite rdfs closure for hundreds of millions of triples. In: 8th International Semantic Web Conference (ISWC2009). (October 2009)
6. Harth, A., Umbrich, J., Hogan, A., Decker, S.: Yars2: A federated repository for querying graph structured data from the web. In: 6th International and 2nd Asian Semantic Web Conference (ISWC2007+ASWC2007). (November 2007) 211–224
7. Aberer, K., Cudré-Mauroux, P., Hauswirth, M., Pelt, T.V.: Gridvine: Building internet-scale semantic overlay networks. The Semantic Web–ISWC 2004 (2004) 107–121
8. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In Elmagarmid, A.K., Agrawal, D., eds.: SIGMOD Conference, ACM (2010) 135–146
9. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank citation ranking: Bringing order to the Web. Technical report, Stanford Digital Library Technologies Project (1998)
10. Bishop, C.M.: Pattern Recognition and Machine Learning (Information Science and Statistics). 1st ed. 2006. corr. 2nd printing edn. Springer (October 2007)
11. Kschischang, F., Frey, B., Loeliger, H.: Factor graphs and the sum-product algorithm. IEEE Transactions on information theory **47**(2) (2001) 498–519
12. Koller, D., Friedman, N.: Probabilistic Graphical Models: Principles and Techniques. MIT Press (Jan 2009)
13. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1973) 235–245
14. Oren, E., Kotoulas, S., Anadiotis, G., Siebes, R., ten Teije, A., van Harmelen, F.: Marvin: Distributed reasoning over large-scale semantic web data. Web Semantics: Science, Services and Agents on the World Wide Web **7**(4) (2009) 305 – 316 Semantic Web challenge 2008.