Semantic Web 0 (0) 1
IOS Press

Signal/Collect

Processing Large Graphs in Seconds

Philip Stutz [a], Daniel Strebel [a], Abraham Bernstein [a]
[a] *University of Zurich*

Abstract.
Both researchers and industry are confronted with the need to process increasingly large amounts of data, much of which has a natural graph representation. Some use MapReduce for scalable processing, but this abstraction is not designed for graphs and has shortcomings when it comes to both iterative and asynchronous processing, which are particularly important for graph algorithms.

This paper presents the Signal/Collect programming model for scalable synchronous and asynchronous graph processing. We show that this abstraction can capture the essence of many algorithms on graphs in a concise and elegant way by giving Signal/Collect adaptations of algorithms that solve tasks as varied as clustering, inferencing, ranking, classification, constraint optimisation, and even query processing. Furthermore, we built and evaluated a parallel and distributed framework that executes algorithms in our programming model. We empirically show that our framework efficiently and scalably parallelises and distributes algorithms that are expressed in the programming model. We also show that asynchronicity can speed up execution times. Our framework computes a high-quality PageRank on a large (>1.4 billion vertices, >6.6 billion edges) real-world webgraph in merely 136 seconds – achieved with only twelve commodity machines.

Keywords: Distributed Computing, Scalability, Programming Abstractions, Programming Models, Graph Processing

1. Introduction

The Web (including the Semantic Web) is full of graphs. Hyperlinks and RDF triples, retweet and social network relationships, citation and trust networks, ratings and reviews – almost every activity on the web is most naturally represented as a graph. Graphs are one of the most versatile data structures. They can be considered a generalisation of other important data structures such as lists and trees. In addition, many structures—be it physical such as transportation networks, social such as friendship networks, or virtual such as computer networks—have natural graph representations. Coupled with the ever expanding amounts of computation and captured data [22], this means that researchers and industry are presented with the opportunity to do increasingly complex processing of growing amounts of graph structured data.

Graph processing means executing an algorithm on graph-structured data. A graph processing program scales if it can take advantage of additional resources. On a commodity cluster architecture, this requires partitioning both the data and computation, such that many computers and processors can execute parts of that computation in parallel. In theory, one could write a scalable program from the ground up for each graph algorithm in order to achieve the maximum amount of parallelism. In practice, however, this requires a lot of effort and is in many cases unnecessary, because many graph algorithms such as PageRank can be decomposed into small iterated computations that each operate on a vertex and its local neighbourhood (or

This paper is a significant extension of [51]. Speficically, it contains a more detailed description of the programming model, describes a larger selection of algorithm adaptations, contains more extensive evaluations, particularly of the distributed version of the underlying framework.

This work is supported by the Hasler Foundation under grant number 11072.

messages from the neighbours). If we can design programming models to express this decomposition and execute the partial computations in parallel on scalable infrastructure, then we can hope to achieve scalability without having to build custom-tailored solutions.

MapReduce is the most popular scalable programming model [9], but has shortcomings with regard to iterated processing [3,12,56,57] and requires clever mappings to support graph algorithms [8,32]. Such limitations of more general programming models have motivated specialised approaches to graph processing [28,38]. Most of these approaches follow the bulk-synchronous parallel (BSP) model [53], where a parallel algorithm is structured as a sequence of computation and communication steps that are separated by global synchronisations. The rigid pattern of bulk operations and synchronisations does not allow for flexible scheduling strategies.

To address the limitations of BSP, researchers have designed programming models for graph processing that are asynchronous [34], allow hierarchical partial synchronisations [30], make synchronisation optional [51], or try to emulate the properties of an asynchronous computation within a synchronous model [54].

Our proposed solution is a vertex-centric programming model and associated implementation for scalable graph processing. It is designed for scaling on the commodity cluster architecture. The core idea lies in the realisation, that many graph algorithms can be decomposed into two operations on a vertex: (1) signaling along edges to inform neighbours about changes in vertex state and (2) collecting the received signals to update the vertex state. Given the two core elements we call our model SIGNAL/COLLECT. The programming model supports both synchronous and asynchronous scheduling of the signal and collect operations.

Such an approach has the advantage that it can be seen like a graph extension of the actors programming approach [21]. Developers can focus on specifying the communication (i.e., graph structure) and the signaling/collecting behavior without worrying about the specifics of resource allocation. Since SIGNAL/COLLECT allows mutiple types of vertices to coexist in a graph the result is a powerful framework for developing graph-centric systems. A foundation especially suitable for Semantic Web applications, as we showed in our development of TripleRush [52] – a high-performance, in-memory triple store implemented with three different vertex types.

We extend our previous work [51] on SIGNAL/COLLECT with a more detailed description of the programming model, a larger selection of algorithm adaptations, a distributed version of the underlying framework, and with more extensive evaluations on larger graphs. Given the above, our contributions are as follows:

- *We designed an expressive programming model for parallel and distributed computations on graphs.*
  We demonstrate its expressiveness by giving implementations of ten algorithms from categories as varied as clustering, inference, ranking, classification, constraint optimisation, and even query processing. The programming model is also modular and composable: Different vertices and edges can be combined in the same graph and reused in different algorithms. Additionally the model supports asynchronous scheduling, dataflow computations, dynamic graph modifications, incremental recomputations, aggregation operations, and automated termination detection. Note that especially the dynamic graph modifications are central for Web of Data applications as they require the seamless integration of ex-ante unknown data.

- *We implemented a scalable open source framework.*
  The framework efficiently and scalably parallelises and distributes algorithms expressed in the programming model. We empirically show that our framework scales to multiple cores and in a distributed setting. We evaluated real-world scalability by computing PageRank on the Yahoo! AltaVista webgraph.[1] The computation on the large (>1.4 billion vertices, >6.6 billion edges) graph took slightly more than two minutes using twelve commodity machines, which is a competitive result both in terms of used resources as well as in terms of computation time.

- *We illustrate the impact of asynchronous algorithm executions.*
  SIGNAL/COLLECT supports both synchronous and asynchronous algorithm executions. We compare the difference in running times between the

[1] Yahoo! Academic Relations, Yahoo! AltaVista Web Page Hyperlink Connectivity Graph, `http://webscope.sandbox.yahoo.com/catalog.php?datatype=g`

asynchronous and synchronous execution mode for different algorithms and problems of varying hardness.

In Section 2 we motivate the programming model and describe the basic approach. We then introduce the SIGNAL/COLLECT programming model in Section 3 and describe our implementation of the model in Section 4. In Section 5 we evaluate both the programming model and the implementation. We continue with a description of related approaches to scalable graph processing and compare them to our approach in Section 6. In Section 7 we examine the limitations of our evaluation and provide an outlook on future work. We finish by sharing our conclusions in Section 8.

2. The SIGNAL/COLLECT Approach: an Intuition

SIGNAL/COLLECT can be understood as a vertex-centric graph processing abstraction akin to Pregel [38]. Another way of looking at it is as an extension of the asynchronous actor model [21], where each vertex represents an actor and edges represent the communication structure between actors. The graph abstraction allows for the composition and evolution of complex systems, by adding and removing vertices and edges. To illustrate this intuition we provide two examples: RDFS sublcass inferencing and the computation of the single source shortest path.

RDFS Subclass Inferencing Consider 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.

Single Source Shortest Path Consider a graph with vertices that represent locations and edges that represent paths between locations. We would like to determine the shortest path from a special location `S` to all the other locations in the graph.
Every location starts out with its state set to the length of the shortest currently known path from `S`. That means, initially, the state of `S` is set to 0 and the states of all the other locations are set to infinity (see Figure 1). In a first step, all edges signal the state of their source location plus the path length (represented by edge weight, in the example above all paths have length 1) to their respective target location. The target locations collect these signals by setting their new state to the lowest signal received (as long as this is smaller than their state). In a second step, the same signal/collect operations get executed using the updated vertex states. By repeating the above steps these operations iteratively compute the lengths of the shortest paths from `S` to all the other locations in the graph.

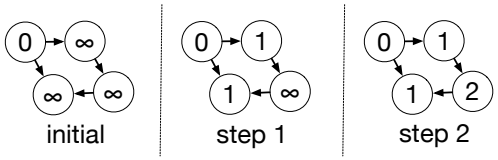


Fig. 1. States of a synchronous single-source shortest path computation with four vertices

In the next section we refine this abstraction to a programming model that allows to concisely express algorithms similar to these examples.

3. The SIGNAL/COLLECT Programming Model

In the SIGNAL/COLLECT programming model 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. Vertices collect the signals and perform some computation on them employing, possibly, some vertex-state, and then signal their neighbors in the compute graph.

3.1. Basic SIGNAL/COLLECT *Graph Structure*

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:

$\mathtt{v}.\mathbf{id}$, a unique id.
$\mathtt{v}.\mathbf{state}$, the current vertex state which represents computational intermediate results.

v.outgoingEdges, a list of all edges e $\in E$ with e.source = 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 neighbour. We 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* 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() method and the default edge type defines an abstract signal() method. To specify an algorithm in the SIGNAL/COLLECT programming model the default types have to be extended with implementations of those method. The collect() method calculates a new vertex state, while the signal() method calculates the signal that is 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.

3.2. The Computation Model and Extensions

In this section we specify how both synchronous and asynchronous computations are executed in the SIGNAL/COLLECT programming model. Also we provide extensions to the core model. The model employs the attribute target on edges to denote the target vertex. Note that all operations pertaining target can be implemented by sending a message to the target vertex simplifying a distributed implementation based on a message bus. We also define two additional methods on all vertices v $\in V$, which enable us to describe computations in SIGNAL/COLLECT:

v.doSignal()
  lastSignalState := state
  for all (e $\in$ outgoingEdges) **do**
    e.target.uncollectedSignals.append(e.signal())
    e.target.signalMap.put(e.sourceId, e.signal())
  end for

v.doCollect()
  state := collect()
  uncollectedSignals := Nil

The doCollect() method updates the vertex state using the algorithm-specific collect() method and resets the uncollectedSignals. The doSignal() method computes the signals for all edges and relays them to the respective target vertices. The additional lastSignalState attribute stores the vertex state at the time of signalling, which is later used for automated convergence detection. Both methods can return values of arbitrary types, which means that vertex states and signals can have arbitrary types as well. With these methods we can describe a synchronous SIGNAL/COLLECT execution.

3.2.1. 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 is executed in parallel, with a global synchronization between the signaling and collecting phases. This parallel programming model is more generally referred to as Bulk Synchronous Parallel (BSP).

Algorithm 1 Synchronous execution

```
for i ← 1..num_iterations do
  for all v ∈ V parallel do
    v.doSignal()
  end for
  for all v ∈ V parallel do
    v.doCollect()
  end for
end for
```

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. We introduce scoring in order to be able to define a computational model that enables us to guide the computation and give priority to more "important" operations.

3.2.2. Extension: Score-Guided Execution

In order to enable the scoring (or prioritizing) of `doSignal()` and `doCollect()` operations, we need to extend the core structures of the SIGNAL/COLLECT programming model and define two additional methods on all vertices $v \in V$:

v.scoreSignal() : Double
: is a method that calculates a number that reflects how important it is for this vertex to signal. A scheduler can assume that the result of this method only changes when the `v.state` changes. The 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 method that calculates a number that reflects how important it is for this vertex to collect. A scheduler can assume that the result of this method only changes when `uncollectedSignals` changes. The 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 methods that capture the algorithm-specific notion of "importance" more accurately, but these methods should not modify the vertex.

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: `s_threshold` and `c_threshold`, which set a minimum level of "importance" for the `doSignal()` and `doCollect()` operations to get scheduled and `max_iter`, which limits the number of computation steps. The algorithm stops when either the maximum number of iterations is reached or all scores are below their thresholds. In the second case we say that the algorithm has converged.

Algorithm 2 Score-guided synchronous execution

```
done := false
iter := 0
while iter < max_iter and !done do
  done := true
  iter := iter +1
  for all v ∈ V parallel do
    if (v.scoreSignal() > s_threshold) then
      done := false
      v.doSignal()
    end if
  end for
  for all v ∈ V parallel do
    if (v.scoreCollect() > c_threshold)
    then
      done := false
      v.doCollect()
    end if
  end for
end while
```

3.2.3. Extension: Asynchronous Execution

We referred to the first scheduling algorithm as synchronous because it guarantees that all vertices are in the same "loop" at the same time. With a synchronous scheduler 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 synchronised.

Asynchronous scheduling removes this constraint: Every vertex can be scheduled out of order and no global ordering is enforced. This means that a scheduler can, for example, propagate information faster by signaling right after collecting. It also simplifies the implementation of the scheduler in a distributed setting, as there is no need for global synchronisation.

Depending on the scheduling strategy an asynchronous scheduler can have the following advantages:

– *Lower latency between operations*
An asynchronous scheduler is not tied to a global ordering that prescribes when information can be propagated. This flexibility can be used to reduce the latency between collecting and signaling and is especially important for use cases such as query processing, where latency is critical (see path query processing 5.1). It can also lead to fewer signal operations due to faster information propagation (see the PageRank analysis in sub-

section 5.3).

– *Reduction of oscillations*
Some synchronous algorithms can be prone to getting trapped in oscillation patterns, where vertices cycle through states in lockstep. Asynchronous processing can reduce such oscillations and allows some of these algorithms to converge quickly (see the vertex coloring evaluation in subsection 5.3).

– *Fewer bottlenecks*
A system that uses asynchronous scheduling can be designed to be more resilient against slow, faulty or overloaded system components.

Algorithm 3 Score-guided asynchronous execution

```
ops := 0
while
ops < max_ops and ∃v ∈ V(
  v.scoreSignal() > s_threshold or
  v.scoreCollect() > c_threshold)
do
  S := choose subset of V
  for all v ∈ S parallel do
    Randomly call either v.doSignal() or
    v.doCollect() iff respective threshold is
    reached; increment ops if an operation was
    executed.
  end for
end while
```

Algorithm 3 shows a score-guided asynchronous execution. Again, three parameters influence when the asynchronous algorithm stops: `s_threshold` and `c_threshold` have the same function as in the synchronous case; `max_ops`, in contrast, limits the number of operations executed instead of the number of iterations. This guarantees that an asynchronous execution either stops 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 to use different scheduling strategies for those operations.

We refer to the scheduler that is we most often use as the "eager" asynchronous scheduler: In order to speed up information propagation this scheduler calls `doSignal` on a vertex immediately after `doCollect`, if the signal score is larger than the threshold.

3.2.4. Distinction: Data-Graph vs. Data-Flow Vertex

When using the synchronous scheduler without scoring (Algorithm 1), then the collect function processes the signals that were sent along all edges during the last signaling step. When we introduce scoring, not all edges might signal during every step. There is a similar issue with asynchronous scheduling: While no signal might be forwarded along some edges, other edges might have forwarded multiple signals. For this reason we distinguish two categories of vertices that differ in the which signals they collect:

Data-Graph Vertex A data-graph vertex is most similar to the behaviour of a vertex in the basic execution mode: It processes `v.signals`, all the values in the signal map. This means that only the most recent signal along each edge is collected. If a multiple signals were received since signals were last collected, then all but the most recent one are never collected. If no signal was sent along an edge, but there was a previous signal along that edge, then this older signal is collected for that edge. This vertex type is suitable for most graph algorithms.

Data-Flow Vertex A data-flow vertex is more similar to an actor. It collects all signals in `v.uncollectedSignals`, which means that it collects all signals that were received since the last collect operation. This vertex type is suitable for asynchronous data-flow processing and some graph algorithms, such as Delta-PageRank (see 5.1).

3.3. Extension: Graph Modifications and Incremental Recomputation

SIGNAL/COLLECT supports graph modifications during a computation. They can be triggered externally or from inside the `doSignal()` and `doCollect()` methods. This means that vertices and edges can dynamically modify the very graph they are a part of. Modifications include adding/removing/modifying vertices/edges or sending signals from outside the graph along virtual edges.

When an edge is added or removed, a scheduler has to update `scoreSignal()` and `scoreCollect()` of the respective vertex in order to check if the modifi-

cation should trigger a recomputation. This is enough to support incremental recomputation for many algorithms and modifications. For some algorithms, however, additional recomputations are also required for vertices when *incoming* edges change. The more powerful incremental recomputation scheme described in [4] could be adapted to cover these cases, but would require an additional extension to track changes of incoming edges.

Modifications are applied in per-source FIFO order, which means that all the modifications that are triggered by the same source are applied in the same order in which they were triggered.

3.4. Extension: Aggregation Operations and Parallel Updates

An aggregation operation over a graph computes a value of an arbitrary type `T` and has two components (see Figure 2).

The `extract()` function is applied to all vertices in the graph (akin to a map function). The `reduce()` function aggregates the extracted values in arbitrary order. Aggregation operations are used to compute global results or to define termination conditions over the entire graph.

Another related feature are parallel updates. A parallel update operation executes an arbitrary function on all vertices of the graph. Potential use cases are submitting results from vertices to a database or normalising a vertex attribute with a global constant that has been computed with an aggregation operation.

The `extract()` function should not modify the vertex. For parallel updates there is a simpler parallel update operation where one can pass a function that can modify vertices.

3.5. Extension: Attributes and Modularity

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, additional attributes and methods.

Inside the same compute graph instances of different vertex/edge types (and, hence, different `signal()` and `collect()` methods) can be combined: Vertices and edges are modular building blocks from which complex heterogeneous graph structures can be built and vertex/edge implementations can be reused across different algorithms.

4. The Signal/Collect Framework — An Implementation

The SIGNAL/COLLECT framework provides a parallel and distributed execution platform for algorithms specified according to the SIGNAL/COLLECT programming model. In this section we explain some interesting aspects of our implementation.

The framework is implemented in Scala, a language that supports both object-oriented and functional programming features and runs on the Java Virtual Machine. We released the framework under the permissive Apache License 2.0[2] and develop the source code publicly, inviting external contributions.

4.1. Architecture

The framework can both parallelise computations on multiple processor cores, as well as distribute computations over a commodity cluster. Internally, the system uses the Akka[3] distributed actor framework for message passing.

A "coordinator actor" bootstraps the "worker actors" and takes care of global concerns such as convergence detection and preventing messaging overload.

The scheduling of operations and message passing is done within workers. Each computer hosts a number of workers and each worker is responsible for a collection of vertices. Workers communicate directly with each other and with the coordinator. Workers have a pluggable scheduler that handles the delivery of signals to vertices and the ordering of signal/collect operations.

Vertices are retrieved from and stored in a pluggable storage module, by default implemented by an in-memory hash map.

A vertex stores its outgoing edges, but neither the vertex nor its outgoing edges have access to the target vertices of the edges. In order to efficiently support parallel and distributed execution, modifications to target vertices from the model are translated to messages that are passed via a message bus.

Every worker and the coordinator have one pluggable message bus each that takes care of sending signals and translating graph modifactions to messages. Optionally it implements features such as bulk-messaging or signal combiners.

[2] `https://github.com/uzh/signal-collect` and `http://www.signalcollect.com`

[3] `http://akka.io/`

`extract(vertex)`	Receives a vertex as input and returns a value of type `T`.
`reduce(values)`	Receives zero, one, or multiple values of type `T` as input and returns one value of type `T`.

Fig. 2. Aggregation operation.

`getWorkerId(vertexId)`	Returns the id of the worker that is responsible for the vertex with id `vertexId`. Default: `vertexId.hashCode mod numberOfWorkers`

Fig. 3. Assignment of vertices to workers.

4.2. Graph Partitioning and Loading

Workers have ids from 0 ascending and by default the graph is partitioned by using a hash function on the vertex ids (see Figure 3). This is similar to how graphs are partitioned in most other graph processing frameworks. For large graphs it usually leads to balanced partitions, but also to a large number of edges between partitions. To improve on this we could adopt some of the optimisations used in the Graph Processing System (GPS) [46]. Because computing a balanced graph partitioning with minimal capacity between partitions is a hard problem itself [1], this would mainly improve performance in cases where algorithms are run on the same graph repeatedly, for long-running algorithms, or when messaging bandwidth is the main bottleneck.

The default storage implementation keeps the vertices in memory for fast read and write access. Extensions for secondary storage can be implemented [50]. Graph loading can be done sequentially from a coordinator actor or preferably in parallel, where multiple workers load parts of the graph at the same time. Specific partitions can be assigned to be loaded by particular workers. This can be used to have each worker load its own partition, which increases the locality of the loading.

4.3. Flexible Tradeoffs

Our framework has defaults that work for a broad range of algorithms, but are not the most efficient solution for most of them. These default implementations can be replaced allowing a graph algorithm developer to choose the trade-off between implementation effort and resulting performance.

An example of a tradeoff is the propagation latency vs. messaging overhead: While sending each signal as soon as possible leads to a low latency and can perform well in local computations, sending each signal by itself will cause a lot of overhead in a distributed setting. Our implementation allows to plug in a custom bulk scheduler and bulk message bus implementation to choose a trade-off that suits the use case (throughput vs. latency).

5. Evaluation

In this section we evaluate the programming model, the scalability of our implementation, and the impact of asynchronous scheduling. The different contributions require different research methods: We evaluate the programming model by adapting important algorithms in a few lines of code. In addition to the expressiveness, we also show that our implementation is able to transparently scale algorithms by empirically measuring the speedup when running algorithms while varying the number of worker threads and cluster nodes. Finally, we compare the impact of asynchronous scheduling versus synchrononous scheduling on different graphs and algorithms.

5.1. Programming Model

One of our main contributions is the simple, compact, yet expressive programming model. Whilst simplicity of a program is difficult to judge objectively, compactness and expressiveness are easier to show.

We demonstrate the expressiveness by giving adaptations of ten algorithms from categories as varied as clustering, inference, ranking, classification, constraint optimisation and even query processing. We show an actual implementation of the PageRank algorithm in Figure 22 in the Appendix. As the example illustrates in comparison to the pseudocode in Figure 4, the pseudocode can be translated to actual code without much boilerplate.

Most algorithms are presented in a simplified version, more advanced versions of many of the examples are available online.[4] To enhance readability and facilitate the comparison of different algorithms, each algorithm is structured in a table representing the three core elements of a computation: The *initial state* represents the state of the vertices when they get added to the graph. The *collect* method uses the vertex state, the appropriate signals for the vertex type, and other vertex attributes/methods to compute a new vertex state. The *signal* method uses attributes/methods defined on the source vertex and edge attributes/methods to compute the signal that is sent along the edge. All described algorithms work on homogeneous graphs that use only one type of vertex/edge, which is specified in the table. Additional information and explanations for complex functions are provided in the algorithm descriptions.

All described algorithms use the default `scoreSignal()` and `scoreCollect()` implementations for automated convergence detection.

PageRank This graph algorithm computes the importance of a vertex based on the link structure of a graph [42]. The vertex state represents the current rank of a vertex (Figure 4). The signals represent the rank transferred from the source vertex to the target vertex. The vertex state is initialised with the `baseRank = 0.15` and the damping factor is usually set to `0.85`. To see how the pseudocode directly maps to an actual algorithm implementation, have a look at an executable PageRank implementation written in Scala in the Appendix (see Figure 22).

It is also possible to implement PageRank as a *dataflow algorithm* by signaling only the rank deltas (Figure 5). This version can be further optimised by not sending the source vertex id with the signals, which saves bandwidth.

Another interesting feature for Linked Open Data is to run an algorithm on data that is dynamically loaded (Figure 6). In order to run PageRank on a dynamically crawled graph, the PageRank vertex loads related triples from the web, adds their objects as new vertices, and adds new edges to connect itself to them. This happens using the graph editor that is available to a vertex inside its `afterInitialisation()` method, which gets called right after the vertex has been added to the graph. This process is repeated recursively until some criterion, for example a maximum crawl depth, is reached.[5]

Single-source shortest path (SSSP) This algorithm computes the shortest distance from one special source vertex (`S`) to all the other vertices in the graph. The vertex states represent the shortest currently known path from `S` to the respective vertex (Figure 7). Edge weights are used to represent distance. The signals represent the total path length of the shortest currently known path from `S` to `edge.target` that passes through `edge`.

Transitive closure One example of computing the transitive closure common to the (Semantic) Web is RDFS subclass inferencing. Each vertex id represents a class and the vertex state represents the set of currently known superclasses of the given vertex. Edges signal the set of currently known superclasses of the class represented by the source vertex (Figure 8). The compute graph is built with the already known (explicit) relationships by adding edges from vertices representing superclasses to vertices representing subclasses.

Vertex Colouring A vertex colouring problem is a special constraint optimisation problem that is solved when each vertex has an assigned colour from a set of colours and no adjacent vertices have the same colour. The following simple and inefficient algorithm solves the vertex colouring problem by initially assigning to each vertex a random colour from some arbitrary set of colours (Figure 9). Then, the vertices check if their own colour (state) is already occupied (contained in the collection of received signals). If such a conflict is encountered, they switch to a random colour except their current colour. The default `scoreSignal()` method ensures that the vertex signals again if there was a conflict. If there was no conflict, then the vertex stays with its current colour.

Algorithms such as this one can solve many optimisation problems such as scheduling or finding solutions for Sudoku puzzles. The described algorithm works with undirected edges. In SIGNAL/COLLECT these are modelled using two directed edges. The performance could be improved by using a better optimi-

[4] `https://github.com/uzh/signal-collect/tree/master/src/test/scala/com/signalcollect/examples`

[5] Code for an executable example can be found at `https://github.com/uzh/signal-collect/blob/master/src/main/scala/com/signalcollect/examples/LodNeighbourhoodPageRank.scala`

`initialState`	`baseRank`
`collect()`	`return baseRank + dampingFactor * sum(signals)`
`signal()`	`return source.state * edge.weight / sum(edgeWeights(source))`

Fig. 4. PageRank (data-graph)

`initialState`	`baseRank`
`collect()`	`return oldState + dampingFactor * sum(uncollectedSignals)`
`signal()`	`stateDelta = source.state - source.signaledState` `return stateDelta * edge.weight / sum(edgeWeights(source))`

Fig. 5. Delta PageRank (data-flow).

`afterInitialisation(editor)`	`triples = loadTriplesFromWeb(id)`
	`objectUrls = extractObjectUrls(triples)`
	`for (url <- objectUrls) {`
	`  editor.addVertex(new LodPageRankVertex(url))`
	`  editor.addEdge(id, new PageRankEdge(url))`
	`}`

Fig. 6. Linked Open Data dynamic PageRank extension.

`initialState`	`if (isSource) 0 else infinity`
`collect()`	`return min(oldState, min(signals))`
`signal()`	`return source.state + edge.weight`

Fig. 7. Single-source shortest path (data-graph/data-flow).

`initialState`	`Set(id)`
`collect()`	`return union(oldState, union(signals))`
`signal()`	`return source.state`

Fig. 8. Transitive closure (data-graph/data-flow).

`initialState`	`randomColour`

`collect()`	`if (contains(signals, oldState))`
	`  return randomColorExcept(oldState)`
	`else`
	`  return oldState`

`signal()`	`return source.state`

Fig. 9. Vertex colouring (data-graph).

sation algorithm of which many have SIGNAL/COLLECT adaptations.[6]

Label Propagation This iterative graph clustering algorithm assigns to each vertex the label that is most common in its neighbourhood [58]. Our variant is called Chinese Whispers Clustering [2] and has applications in natural language processing. The algorithm works on graphs with undirected edges which are modelled with two directed edges.

The vertex state represents the current vertex label (= cluster) and it is initialised with the vertex id (Figure 10). This means that each vertex starts in its own cluster. Then, labels are propagated to neighbours. When a vertex receives neighbours' labels, it appends its own label to the collection of labels signalled by the neighbours. It then updates its own label to the most frequent label in that extended collection. Ties are broken arbitrarily.

Relational Classifier Relational classification can be considered a generalisation of label propagation. The presented classifier (Figure 11) is a variation of the probabilistic relational-neighbour classifier described by Macskassy and Provost [36,37]. The algorithm works on graphs with undirected edges which are modelled with two directed edges. `ProbDist` represents a probability distribution over different classifications. Each vertex starts with an initial probability distribution over the classes, which can be uniform or can reflect the observed frequencies of classes in the training set. If the class of a vertex is available as training data, then that class gets probability 1 and is not changed by the algorithm. When a vertex receives the distributions of its neighbours, then it updates its own distribution to a normalised sum of the class probability distributions of the neighbours. A collective inference scheduling can be chosen using the `scoreSignal()` implementation to determine when a vertex informs its neighbours about its label distribution. This classifier only works when edges are more likely to connect vertices that belong to the same class (homophily) and, given its supervised nature, when some classes are known as training data [36]. The algorithm described here can be extended with a local classifier to determine the initial state. This initial state would be added to the sum of neighbour states in the `collect` method (potentially with a higher weight). In this case, it is no longer necessary for some classes to be known. The homophily constraint could be dropped by using a more advanced relational classification algorithm [5,14].

Conway's Game of Life (Life) Life is played on a large checkerboard of cells, where each cell can be in one of two states (dead/alive) [13]. The game progresses in turns and each turn the state of a cell is updated based on the states of its neighbouring cells. The game is mapped to SIGNAL/COLLECT by representing each cell as a vertex, alive is represented with state 1 and dead as state 0 (Figure 12). The neighbourhood relationships between the cells are modeled with edges between neighbouring cells. A vertex counts how many of its neighbours are alive and uses this to determine its state next turn accroding to the rules of the game. Life is famous for the complex patterns that can emerge from its simple rules.

Threshold Models of Collective Behaviour Threshold models of collective behaviour are used to model situations in which agents have two options and the risk/payoff of each option depends on the behaviour of neighbouring agents [16]. The risk/payoff is determined by a threshold which can be different for every actor. Threshold models allow to model the collective behaviour of a group of actors. Granovetter uses the example of rioting, but argues that such models can also be used to model innovation, rumor diffusion, disease spreading, strikes, voting, migration, educational attainment, and attendance of social events.

Such models can be mapped to SIGNAL/COLLECT by representing each agent with a vertex and connecting all agents that can observe each other's behaviour with edges. The initial state determines the default behaviour of an agent (Figure 13). In our example, an actor does not riot initially, unless it is a natural rioter, which means that the agent would riot no matter what its neighbours do. Edges inform neighbours about the behaviour of an agent and in the `collect()` method the agent analyses what fraction of its neighbours are taking the alternative decision. If the fraction of neighbouring agents that display a behaviour exceeds the individual threshold (in our example `riotingThreshold`), then the agent changes its behaviour and switches to the alternative behaviour (e.g., start to riot).

[6] A Distributed Stochastic Algorithm implementation in SIGNAL/COLLECT, for example, can be found at: `https://github.com/elaverman/signal-collect-dcops/blob/2e25766c04d66a6cdce4ec5a659fb0dfc45436d6/src/main/scala/com/signalcollect/approx/flood/DSA.scala`

initialState	id
collect()	return mostFrequentValue(append(oldState, signals))
signal()	return source.state

Fig. 10. Label propagation (data-graph).

initialState	if (isTrainingData) trainingData else avgProbDist

collect()	if (isTrainingData)
	  return oldState
	else
	  return signals.sum.normalise

signal()	return source.state

Fig. 11. Relational classifier (data-graph).

initialState	if (isInitiallyAlive) 1 else 0

collect()	switch (sum(signals))
	  case 0: return 0 // dies of loneliness
	  case 1: return 0 // dies of loneliness
	  case 2: return oldState // same as before
	  case 3: return 1 // becomes alive if dead
	  other: return 0 // dies of overcrowding

signal()	return source.state

Fig. 12. Conway's Game of Life (data-graph).

initialState	if (isNaturalRioter) true else false

collect()	riotingNeighbours = filterTrue(signals)
	rioterFraction = riotingNeighbours.size / signals.size
	if (rioterFraction > riotingThreshold)
	  return true
	else
	  return false

signal()	return source.state

Fig. 13. Threshold models of collective behaviour (data-graph).

Matching Path Queries This algorithm matches path queries, which is a typical use case for a graph processing system. The signals sent in the algorithm initially come from outside the graph along a virtual edge. The signals are path queries that specify a pattern of vertices and edges that they can match. An example for such a pattern might be: Match any path that starts with a vertex that has a "professor" property, continues along an edge that has an "advises" property and ends with an arbitrary vertex.

Once a query arrives at a vertex, its first part is matched with the vertex at which it has arrived (Figure 14). This is done with the `successfulMatchesWithVertex()` function, which returns only the queries that have successfully matched with the local vertex.

If the query is fully matched—meaning all parts of its path are bound to a vertex or edge—then this path is reported as a result (this could be done by adding it to some result attribute that is later picked up by an aggregation operation). If there is still a part of the query left that needs to be matched, then it is added to the state set of the vertex. During the signal operation all edges try to match the next part of the queries—the one potentially constraining the type of edge to follow—using their `successfulMatchesWithEdge()` func-

tions. Queries that were successfully edge-matched are returned by that function and signalled along the respective edges.

Matching path queries has many use cases: From simpler ones such as triangle/cycle detection, the approach could be extended to more complex tasks such as computing random walks with restarts or even matching expressive graph query languages.

Artificial Neural Networks Artificial neural networks are the result of an attempt to imitate the structure of biological neural networks and there are *"literally tens of thousands of published applications"* [45, p. 748]. Neural networks consist of nodes connected by links [45, p. 737]. The nodes are mapped to vertices in SIGNAL/COLLECT and the links are mapped to directed edges. Activations are sent as signals between edges, with the difference that they already get adjusted for edge weight in the `signal()` method of the edge (Figure 15). The activation function is mapped to the `collect()` method and updates the state that represents the unit activation. Varying inputs are sent from outside the graph as signals along virtual edges.

Sketching of some additional algorithms The "Bipartite Matching" and "Semi-Clustering" algorithms described in the Pregel paper [38] can be adapted to the SIGNAL/COLLECT model by separating the compute function into a signal part for communication and a collect part for the state update. They require access to the step number, which is not available in the default SIGNAL/COLLECT model, but can be added for example by using parallel update operations between computation steps. An adaptation of "Loopy Belief Propagation" has been outlined in [51] and was used to do inference on Markov logic networks [47].

The presented algorithms were chosen because of their natural fit to the model. Many more algorithms can be mapped by using multiple vertex and edge types in the same computation, by allowing for complex computation in the vertices/edges or by interleaving iterations with aggregation operations.

We have also implemented a triple store with competitive performance inside SIGNAL/COLLECT [52]. This system uses three different vertex types to model an index and to keep track of query executions.

The main benefit of adapting an algorithm or system to the SIGNAL/COLLECT model is that its execution is automatically scheduled in parallel (or even distributed, if several machines are available), which allows for scalability. In the next section we empirically evaluate the scalability of our framework when executing algorithms expressed in the programming model.

5.2. Scalability

In this subsection we evaluate the scalability of the SIGNAL/COLLECT framework. For the framework to be scalable, it needs to be able to use additional resources to speed up algorithm executions. In order to evaluate this we empirically measure the performance of our framework on multiple algorithms while varying the available resources. More specifically, we analyse the scalability by varying (1) the number of worker threads and (2) the number of cluster nodes.

5.2.1. Multi-core (vertically/scale up)

We determined the multi-core scalability by measuring the parallel speedup when running an algorithm on the same graph, but with a varying number of worker threads.

In a first benchmark we ran the SSSP and PageRank algorithms on a machine with two twelve-core AMD Opteron™ 6174 processors and 66 GB RAM. We executed these algorithms with both synchronous and "eager" asynchronous scheduling. They were run on the web graph dataset[7] with 875 713 vertices (websites) and 5 105 039 edges (hyperlinks). Each combination of algorithm and scheduler was run whilst varying between 1 and 24 worker threads (as the machine has 24 cores). Each run was executed ten times and we graphed the resulting average running time in Figures 16 (PageRank) and 17 (SSSP), where the error bars indicate min/max running times. The speedup was calculated relative to the average runtime with one worker thread. Each execution was run cold in a new JVM, because this reflects the actual usage best.

PageRank was run with a signal function that returns the delta between the previous signal state and the current state (see Figure 5). The signal threshold was set to 0.01, which determines the precision of the result. More detailed evaluation parameters can be found in the evaluation program.[8]

[7] `http://snap.stanford.edu/data/web-Google.html`

[8] The evaluation program used was MulticoreScalabilityEvaluation in `https://github.com/uzh/signal-collect-evaluation` at revision 05057d000d. The snapshot dependencies were `https://github.com/uzh/signal-collect` at revision ba26e95e20 and `https://github.com/uzh/signal-collect-graphs` at revision 0149927e68. The results are available at

`initialState`	`emptySet`

`collect()`	`matched = successfulMatchesWithVertex(signals)`
	`(fullyMatched, partiallyMatched) = partition(matched)`
	`reportResults(fullyMatched)`
	`return union(oldState - lastSignalState, partiallyMatched)`

`signal()`	`return successfulMatchesWithEdge(source.state)`

Fig. 14. Matching path queries (data-flow).

`initialState`	`0`
`collect()`	`return 1 / (1 +` $e^{-\texttt{signals.sum}}$`)`
`signal()`	`return source.state * edge.weight`

Fig. 15. Artificial neural networks (data-graph).

The fastest running time was 7 seconds for PageRank and 1.2 seconds for SSSP. The speedup when going from 1 core to 24 cores was 9 for SSSP and around 13 for PageRank. This shows that SIGNAL/COLLECT scales with additional cores, but that the actual factor depends on the algorithm. The achievable speedup also depends on the graph structure: Running SSSP on a chain of vertices would not allow for any parallelism with our implementation.

5.2.2. Distribution (horizontally/scale out)

We determined the distributed scalability by measuring the speedup when running an algorithm on the same graph, but with a varying number of cluster nodes.

Whilst the previous benchmarks ran with a simple PageRank implementation, we ran an optimised version of the Delta PageRank algorithm on the Yahoo! AltaVista webgraph[9] with 1 413 511 390 vertices and 6 636 600 779 edges. This is one of the largest real-world graphs available for such evaluations and a realistic use case for the PageRank algorithm. In order to measure scalability we ran the algorithm ten times with each configuration on a cluster with 4, 6, 8, 10, and 12 nodes, with 24 worker threads per node (i.e., 96 workers up to 288 workers). The nodes were the same 24-core machines used in the multi-core scalability evaluation connected by a 1 gigabit ethernet switch. In all computations the coordinator actor was running on a laptop on a different network, this machine had no problem handling the load of running termination detection and flow control. The latency between coordinator and workers was less than one millisecond. Given that the heartbeat interval was 100 milliseconds, it is unlikely for the remoteness of the coordinator to have had much effect on the computation.

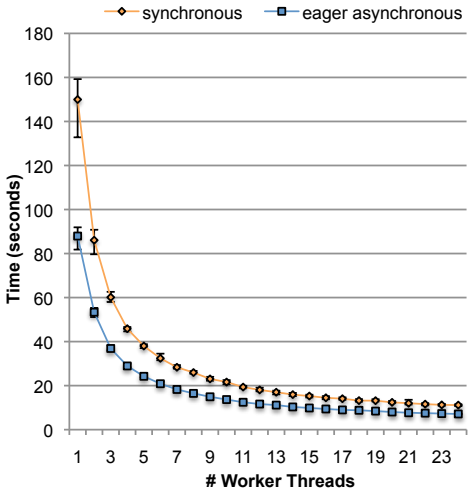
The vertices were partitioned using hashing as described earlier, so most edges spanned different workers and nodes. The graph was loaded from the local file system of the machines and loading took between 45 seconds (fastest run with 12 nodes) and 235 seconds (slowest run with 4 nodes).

The huge amount of signals required more efficient usage of bandwidth, which is why we used a bulk scheduler and bulk message bus. When scaling across more nodes and workers, this means that either each bulk signal has to contain fewer signals or that there is increased latency that would impair algorithm convergence. We chose to keep the latency constant, which has the effect of reducing the benefits of bulk signaling for runs with more nodes, but removes convergence characteristics as a confounding factor.

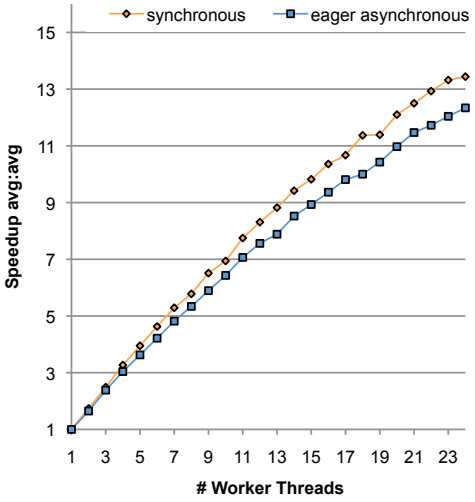
Another optimisation that we used was to have the vertex change the edge representation and for each edge only store the ID of the target vertex. The target IDs are integers, so to further reduce the memory footprint we sorted the array of target IDs and at each position only stored the delta from the previous array entry. We then took advantage of the smaller IDs by using variable length encoding on the ID deltas. Furthermore, we collected signals right when they were delivered, which makes it unnecessary to store them inside the vertex until they are collected. These optimisations

https://docs.google.com/spreadsheet/ccc?key=0AiDJBXePHqCldEVwYk1lWDJpQmVRc0QtUWxLcFVXUWc.

[9] Yahoo! Academic Relations, Yahoo! AltaVista Web Page Hyperlink Connectivity Graph, http://webscope.sandbox.yahoo.com/catalog.php?datatype=g

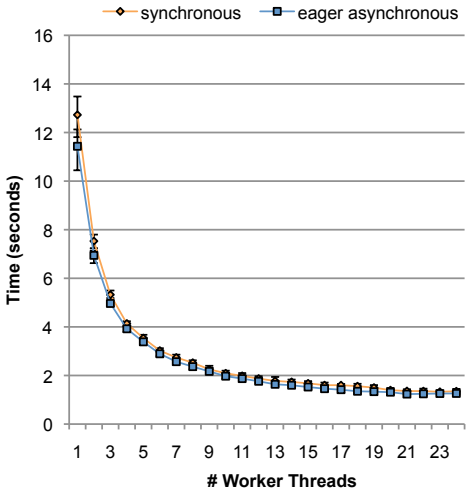


(a) Execution Times

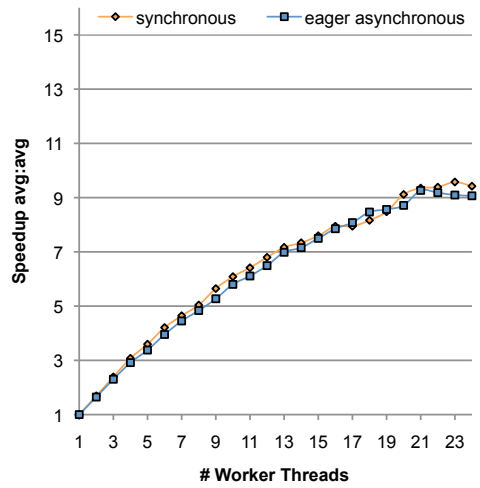


(b) Speedup

Fig. 16. Multi-Core Scalability of PageRank



(a) Execution Times



(b) Speedup

Fig. 17. Multi-Core Scalability of Single-Source Shortest Path

reduced the memory footprint and allowed us to successfully run the algorithm on only four machines[10].

Finally, Delta PageRank was run with a signal function that returns the delta between the previous signal state and the current state. Every execution ran until convergence with a signal threshold of 0.01 and both the vertex state (PageRank) as well as the signals were represented as floating point numbers. The state was not updated if adding the damped signal would have increased the state by more than if the full signal had been added. This additional safeguard is necessary to guarantee convergence.

Figure 18(a) graphs the execution times. It shows that increasing the number of nodes decreases runtime. Indeed, the speedup plot in Figure 18(b) shows that for using three times more resources we get a speedup of almost two. This is decent, considering that more nodes means a larger fraction of signals are sent across nodes (over the slow network, as opposed to fast in-memory transfers) and that there is more overhead for signaling due to smaller bulk signal sizes. More de-

[10]See `https://github.com/uzh/signal-collect-evaluation/blob/master/src/main/scala/com/signalcollect/evaluation/algorithms/MemoryMinimalPage.scala` for the 60-line custom vertex implementation

tailed evaluation parameters can be found in the evaluation program.[11]

Comparison with other systems We are well aware that comparing run-times between systems run on different machines is a problematic proposition at best. The main goal of the comparisons below is, therefore, to provide an intuition of the order of scalability and performance of SIGNAL/COLLECT in contrast to other systems currently reported on in the literature.

Pegasus [27] is a MapReduce-based system that ran 10 iterations of an iterative belief propagation algorithm on the same Yahoo! Altavista webgraph using 100 machines of a Hadoop cluster. This computation took 4 hours.

GPS [46] computed 50 iterations of PageRank on a webgraph with 51 million vertices and 1.9 billion edges in *846 seconds* using a cluster of 60 Amazon EC2 nodes (4 virtual cores and 7.5GB of RAM each). Using a pre-partitioned graph reduced the computation time to 372 seconds. In their evaluations they describe that GPS runs more than an order of magnitude faster than Giraph.

GraphLab did not report any evaluations for the PageRank algorithm, which complicates comparison. The largest (pre-partitioned) graph it was evaluated on had 27 million vertices and 375 million edges. The non-partitioned ones were smaller.[12]

PowerGraph [15] required about 14 seconds to compute PageRank on a Twitter follower graph with *40 million vertices* and 1.5 billion edges employing a cluster of 64 Amazon EC2 nodes (8 cores and 23GB of RAM each, connected by 10 gigabit ethernet). They report faster times with coordinated partitioning requiring an up-front loading time of more than 200 seconds. In their evaluations PowerGraph is at least an order of magnitude faster than the other frameworks they compare against.

[11] The evaluation program used was DistributedWebGraphScalabilityEval in `https://github.com/uzh/signal-collect-evaluation` at revision 701e208. The snapshot dependencies were `https://github.com/uzh/signal-collect` at revision 43c3b0ffe7 and `https://github.com/uzh/signal-collect-graphs` at revision f35637c930. The results are available at `https://docs.google.com/spreadsheet/ccc?key=0AiDJBXePHqCldDF2dEJIWnliVkJ0cjBrVlVvOTBkMkE`.

[12] GraphLab did not scale up to the Yahoo! Altavista webgraph according to the thesis defence slides of Joseph E. Gonzalez, slide 70, `http://www.cs.cmu.edu/~jegonzal/talks/jegonzal_thesis_defense.pptx`.

PageRank quality To exclude the possibility that SIGNAL/COLLECT might be so fast because it computes a low-quality PageRank we reran the evaluation with 12 nodes using double precision signals and vertex states. The signal threshold for this algorithm determines how much a PageRank score has to change in order to trigger another signal operation, so we also varied it by several orders of magnitudes down to 0.000001, which should return very precise results. For each precision level we used ten runs, error bars indicate min/max running time (see Figure 19). The highest level of precision increased the average running time by almost a factor of five compared to the running time in the distributed scalability evaluation.

To determine the quality of the PageRanks that we computed in the distributed evaluation in a bit more than 2 minutes (Figure 18) we compared the result with a PageRank that was compute with double precision signals and states, as well as a much smaller signal threshold (double precision, signal threshold of 1E-7 in Figure 19).

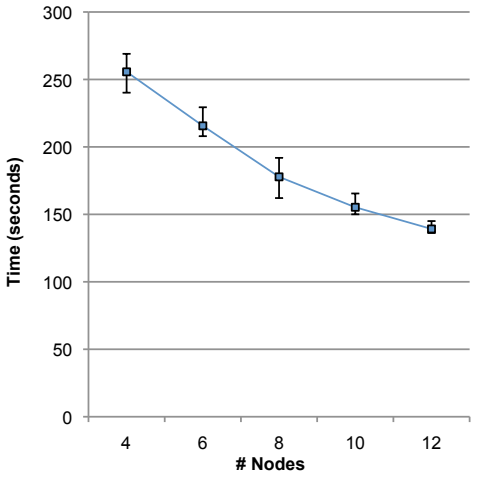
We computed the Spearman's rank coefficient on the 1000 top ranked vertices. The correlation coefficient between the rankings is an almost perfect $\varrho_s = 0.999$. This means that for all practical ranking purposes the PageRank from our distributed scalability evaluation should be precise enough.

5.3. Synchronous vs. Asynchronous

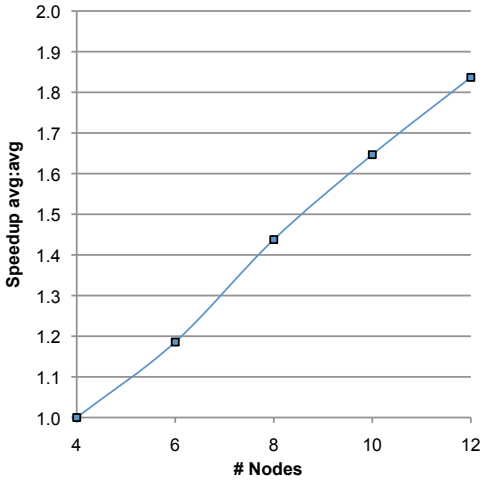
In subsection 3.2.3 we described the potential benefits of asynchronous scheduling. In order to measure how much impact the lower latency signal propagation has we ran PageRank and SSSP with both kinds of schedulers in the previously described scalability experiments. As reported in Figures 16 and 17 asynchronous scheduling is on average between 36% (with 24 workers) and 41% (with 1 worker) faster than synchronous scheduling. This is largely because of earlier signal propagation: In all cases the asynchronous version required on average 30% fewer signal operations until convergence.

Scheduling does not have the same impact on all algorithms: The single-source shortest path algorithm took approximately the same amount of time, regardless of the scheduling.

To evaluate the *impact of scheduling on oscillations* we ran a greedy algorithm to solve vertex colouring



(a) Execution Times



(b) Speedup

Fig. 18. Horizontal scalability of PageRank on the Yahoo! AltaVista Web Page Hyperlink Connectivity Graph with 1 413 511 390 vertices and 6 636 600 779 edges. The data points in 18(a) show the average execution time over 10 runs and the error bars indicate the fastest and slowest runs. 18(b) plots the speedup relative to the average execution time with 4 nodes. The signal threshold used was 0.01, state and signals were represented as floats.

problems on the Latin Square dataset.[13] The graph is a vertex matrix with 100 columns and 100 rows, where all vertices in each column and all vertices in each row are connected (modeled by almost 2 million undirected edges in SIGNAL/COLLECT). The problem requires

[13]We used the dataset provided by CMU at `http://mat.gsia.cmu.edu/COLOR04/INSTANCES/qg.order100.col`

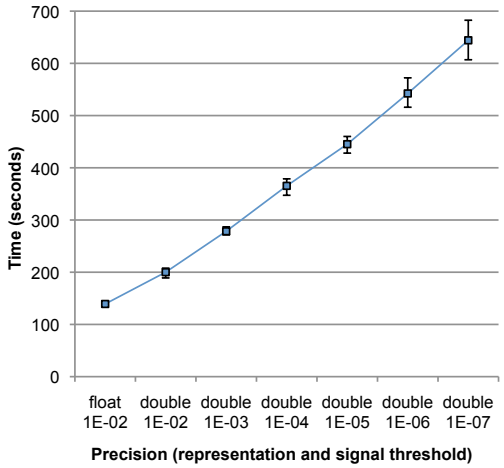


Fig. 19. PageRank on the AltaVista Web Graph using 12 nodes with varying signal/state representations and signal thresholds. The data points show the average execution time over 10 runs, while the error bars indicate the fastest and slowest runs.

at least 100 colours to be solved and becomes easier to solve when more colours are available. We ran the algorithm with both synchronous and "eager asynchronous" scheduling for a varying number of available colours. The hardware used was the same as in subsection 5.2.1. For each number of colours there were ten executions with 24 workers. In Figure 20 we show the fastest run of the ten for each number of colours. Executions were terminated after 20 minutes, even if no solution was found[14]. Each vertex was initialised with the same colour, initialising with random colours would lead to faster executions.

Figure 20 shows the executions with "eager" asynchronous scheduling found solutions much quicker than synchronous executions. For the harder problems with fewer colours there are also several cases where a synchronous scheduling fails to find a solution within the time limit, while the asynchronous scheduling found a solution within a few seconds. One explanation is that with a synchronous scheduling the vertices tend to switch states in lockstep, which has them

[14]The evaluation program used was VertexColoringSyncVsAsyncEvaluation in `https://github.com/uzh/signal-collect-evaluation` at revision f53d9897b1. The snapshot dependencies were `https://github.com/uzh/signal-collect` at revision 40d89ba1c1 and `https://github.com/uzh/signal-collect-graphs` at revision 0149927e68. The results are available at `https://docs.google.com/spreadsheet/ccc?key=0AiDJBXePHqCldEFORUhISkJVSy15Nmd6QnlqYzFKWUE`.

cycle through or oscillate between conflicts ("thrashing"). The results show that for some algorithms asynchronous scheduling can be crucial for fast convergence. 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 in that context that [29, p. 408]: *"In practice an asynchronous message passing scheduling works significantly better than the synchronous approach."*

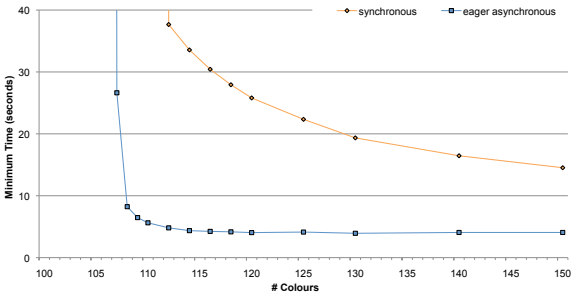


Fig. 20. Vertex colouring with a varying number of colours on a Latin Square problem with 100 * 100 vertices and almost two million directed edges.

6. Related Work

In this section we give an overview over the foundations of SIGNAL/COLLECT and alternative approaches to large-scale graph processing.

6.1. Foundations

The SIGNAL/COLLECT programming model is related to three lower-level programming models, which are all suitable for distributed and parallel computations, but lack a focus on graph computations:

- *Bulk-synchronous parallel (BSP) [53]*
  In BSP, a parallel computation consists of a sequence of supersteps. During each superstep, components process some assigned task and communicate with each other. There is a periodic global synchronization that ensures that all tasks of the superstep have been completed before the next superstep is started. The synchronous scheduler of SIGNAL/COLLECT is similar to this model.

- *Actor model [21]*
  In the actor programming model, many processing components take part in a computation and operate in parallel. These components can only influence each other via asynchronous message passing. The asynchronous scheduler of SIGNAL/COLLECT was inspired by the actor model.

- *Data-flow model*
  Depending on the context, the expression "dataflow" can have different meanings. We understand it broadly as a programming model where a computation is defined as a dependency graph in which data flows along edges and vertices use their input data to compute new data that gets sent along their outgoing edges. This model can be seen as a specialisation of the actor model, where each vertex is represented by an actor and communicates along the graph structure.

When designing a programming model for graph processing, it is important to consider the different kinds of computations on graphs. There are two fundamentally different ways of thinking about computations on graphs. One way is to interpret a graph is as a data structure, where data can be associated with vertices and edges. Computations may explore this structure and modify its data, potentially iteratively, until some termination or convergence criterium is reached. We refer to computations with this characteristic as *data-graph* computations. Another way to interpret a graph is as a plan that determines the flow of data along processing stages. Vertices represent processing stages for data, while edges represent the (potentially cyclic) paths along which data flows. This view encompasses the *data-flow* programming model.

With SIGNAL/COLLECT, we have designed a programming model that is suitable for both kinds of computations: In the SIGNAL/COLLECT programming model vertices are processing units akin to actors whilst edges can have associated data and may compute signals that flow to their target vertex.

Researchers with roots in disparate communities such as machine learning, biology, or the semantic web have answered the call for general programming models and frameworks specialised for scalable graph processing.

Figure 21 provides a high-level overview of distributed data processing systems that support iterated processing. It differentiates systems along their abil-

ity for synchronous versus asynchronous processing of the data on the y-axis and the kind of data abstraction they operate on (key-value pairs, sets, or tables versus graphs) on the x-axis. The category "Synchronisation Required" encompasses systems that schedule iterated computations with mandatory global barriers between iterations. The "Asynchronous Possible" category encompasses systems that are able to schedule iterated computation without such global barriers. "MR Based" is used as a category label for systems that extend the MapReduce model with support for iterated processing or graph abstractions.

We focus our discussion on programming models and systems that are geared towards processing graphs, especially ones that are capable of asynchronous processing. Specifically, we first present GraphStep and Pregel, which have inspired many other graph specialised BSP-based systems. After that, we discuss GraphLab, its extension PowerGraph, and HipG in detail, because they are vertex centric approaches that are closely related to SIGNAL/COLLECT. In the last part we give summaries of other related work.

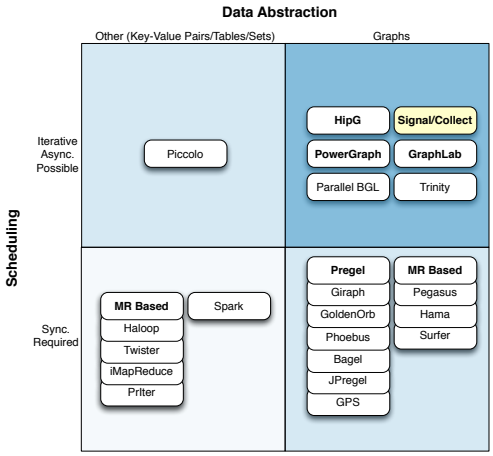


Fig. 21. **Selected Related Work:** An overview of only the high-level distributed data processing *systems* that support iterated processing.

6.2. GraphStep and Pregel

GraphStep [11] is the programming model that is most closely related to SIGNAL/COLLECT. It combines the BSP programming model with the concept of vertices as actors and edges representing the communication structure between those actors. A computation progresses with all vertices receiving input messages, awaiting a global synchronization, performing a local update operation, and sending output messages. The last two steps broadly correspond to collecting and then signaling. A newer description of the model [10] adds a reduce phase on the incoming messages and a separate edge function where each edge reads and writes local state and then possibly sends a message to its destination vertex. The model is meant to be implemented in field-programmable gate array (FPGA) circuits and does not support data-flow computations, asynchronicity, or graph modifications.

Pregel is a framework with a similar programming model developed by Google for large-scale graph processing [38]. The framework scales to graphs with billions of vertices and edges via distribution to thousands of commodity PCs. Pregel is based on a programming model that was inspired by BSP: A computation consists of a sequence of supersteps. During a superstep, each vertex executes an algorithm-specific compute function that can modify the vertex state, modify the graph, and send messages to other vertices. Global synchronisations ensure that all compute functions of the superstep have been completed before the next superstep is started. Within a compute function, a vertex can vote to halt the computation. A computation ends when all the vertices have voted to halt. In order to reduce the number of messages that are sent between workers/machines, Pregel supports combiners that aggregate multiple messages for the same vertex into one. For the computation of global values Pregel also supports aggregation operations.

The Pregel model merges computation, communication, and termination detection into one compute function on a vertex. This function is a black box from the perspective of the framework, which requires a manual implementation of termination detection and prevents the scheduler from separately scheduling state updates and communication.

Pregel can only handle synchronous computations. In a synchronous computation one problematic operation or node can be enough to slow all computations, while in an asynchronous computation only operations on that node or the specific operation are slowed. As we discussed in our evaluation (see Section 5.3), synchronous scheduling can also lead to convergence problems due to oscillations for some algorithms.

Pregel supports graphs with one kind of vertex type sharing a single compute function.This complicates the reusability of vertex/edge-specifications and it adds complexity when implementing algorithms with multiple kinds of vertices or edges. These constraints also

make it harder to compose several algorithms within the same computation.

There are extensions to the model for incremental recomputations [4] and for custom scheduling that can imitate some of the properties of an asynchronous scheduling [54].

Google has not released its implementation of Pregel, but there exist several related open source implementations such as Giraph[15], GoldenOrb[16], Phoebus[17], Bagel[18], and JPregel[19]. In addition, two Pregel-like implementations were developed with a special focus: Menthor[20] is an open source implementation associated with research into high-level control structures over computation steps [20] and the Graph Processing System (GPS) implementation supports static and dynamic graph partitioning [46]. Also noteworthy is Green-Marl [23], a domain-specific programming language for graph analysis algorithms that compiles to execution systems with a Pregel-like programming model.

6.3. GraphLab

GraphLab is a programming model and framework for parallel graph algorithms [34]. The programming model is especially suitable for computations with sparse data dependencies and for asynchronous iterative computation.

GraphLab is based on a data-graph model which simultaneously represents data and computational dependencies. A computation progresses by executing update functions on vertices. These functions can modify the vertex and edge data as well as data associated with neighbouring vertices in the data-graph. The model offers flexible scheduling of these update operations as well as functions to aggregate over the state of the entire data-graph. The scheduler supports different consistency guarantees, which permit the adaptation of some algorithmic correctness proofs from a sequential to a parallel setting.

In full-consistency mode, concurrent modifications to the neighbourhood of a vertex have to be prevented while an update function is executed. Assuming a random distribution of vertices over cluster nodes of a large cluster—an assumption currently true for many frameworks such as HipG (see below) and Pregel—the expected (and worst-case) scenario is that the vertex data and the data of neighbouring vertices are spread over almost as many nodes as there are vertices in the neighbourhood. The authors describe in a more recent publication [15] (see below) that executing an update function in full-consistency mode on a vertex with a sizable neighbourhood is a costly operation, because it requires a distributed lock of a large fraction of the cluster. They also mention that "distributed locking and synchronization introduces substantial latency" [33]. Furthermore, they describe that "the locking scheme used by GraphLab is unfair to high degree vertices" (see [15], Section 4.3.2).

Another scheduler ("chromatic engine") works around some of these issues in distributed computations [33]. This scheduler uses a vertex colouring to avoid the expensive locking during execution. It is equivalent to a BSP execution where at each step only vertices with the same colour are active. This scheduler requires finding a graph colouring (more constrained ones for strong consistency guarantees) and the number of processing steps and global synchronisations is multiplied by the number of colours used for the graph colouring. There seems to be a trade-off between the availability of consistency guarantees and the effort obtaining a graph colouring.

GraphLab does currently not allow graph modifications during a computation and does not support graphs with multiple vertex types, which complicates composition of algorithms and reusability of components. Lastly, GraphLab has undirected edges. Hence, algorithms that exploit directionality would have to encode it in an edge's data, complicating the framework's ability to optimise computations based on directionality.

6.4. PowerGraph

PowerGraph [15] is a substantial redesign and reimplementation of GraphLab. The main difference in PowerGraph's abstraction lies in the computation's division into three phases: gather roughly corresponds to a Pregel combiner gathering and aggregating the data from neighbours, apply computes the new vertex state, and scatter updates the values of the adjacent edges. According to the execution semantics ([15], Algorithm 1, Section 4.1) the three functions are always called sequentially, without interruption, possi-

[15] http://giraph.apache.org/
[16] http://goldenorbos.org/
[17] https://github.com/xslogic/phoebus
[18] https://github.com/mesos/spark/blob/master/bagel
[19] http://kowshik.github.com/JPregel/
[20] http://lcavwww.epfl.ch/~hmiller/menthor/

bly complicating scheduler-based optimisations. Akin to GraphLab, PowerGraph does not allow for multiple implementations of the three functions per graph and it does not support graph modifications during computations.

To enable a more efficient implementation of the distribution PowerGraph introduces the idea of vertex cuts – essentially the replication of vertices to many machines. Whilst this reduces cross-machine communication for some algorithms it introduces replication of the vertices and their associated state/data up to a factor of 5-18 for 64 machines. The variation of the overhead factor is dependent on the partitioning strategy chosen. Smarter strategies reduce the replication factor but increase graph loading time by a factor of about 5 (when using 64 machines).

6.5. HipG

HipG is a distributed framework that facilitates high-level programming of parallel graph algorithms by expressing them as a hierarchy of distributed computations [30,31].

As in the Pregel model, code is executed on a vertex. But while in Pregel messages are sent to other vertices, a HipG vertex can conceptually directly execute functions on neighbouring vertices (the framework translates those function calls to asynchronous messages). HipG supports synchronisers which are coordinators for function executions that have the option to block until all executed functions have completed. This feature can also be used to aggregate global values. A synchroniser can spawn additional synchronisers to create hierarchical computations. This is especially useful for divide and conquer algorithms on graphs.

While it is possible to write a compute function for a vertex that handles thousands of received messages at once, there is no obvious way of combining functions, which means that they all have to be executed. This could be problematic if one wants to implement an iterated computation, because it would require for a function to spawn as many new functions as there are neighbours, potentially leading to an exponential growth of functions in the system. One solution is to use a global synchroniser that repeatedly executes functions on all vertices (using a "visit" flag and only propagating onwards if the flag is not set yet) and has barriers (synchronisations) between those executions (indeed, this is how the PageRank example is implemented in the example code provided with the system[21]) – an implementation of a BSP-scheduler with HipG primitives.

6.6. Other Related Work

The Parallel BGL[22] is a generic C++ library of parallel and distributed graph algorithms and data structures [17,18,35]. One of the main design goals of this system (and also of the ParGraph[23] system) was to allow for sequential BGL[24] algorithms to be "lifted" to parallel programs whilst minimising the required changes. It has support for a special process group that delivers messages immediately instead of waiting for a BSP step synchronisation, but this feature is not explored in any depth.

Najork et al. [40] evaluated three different platforms and programming models on large graphs. The focus of the evaluation is to find the trade-offs between the platforms for various algorithm. The evaluation did not include vertex-centric models. Members of the same lab are also working on the Trinity graph engine, a distributed key-value store with optimisations for vertex-centric graph processing, such as bundling of messages, graph partitioning and low latency processing [49]. Trinity also supports asynchronous processing and while the technology and features of the framework are impressive, it is not publicly available and the report gives little information about the properties of the programming model and about how algorithms are expressed.

There are several systems for large-scale graph computations implemented on top of MapReduce or by generalising the MapReduce model. Most of these systems have limited support for iterated computations and do not support asynchronicity [7,19,28,48]. PrIter is a modified version of Hadoop MapReduce that supports executing processing steps only on a subset of items with priorities above a threshold [57]. Kajdanowicz et al. [26] compared the efficiency of a MapReduce-based system with a BSP-based system for processing large graphs and conclude that BSP can outperform MapReduce by up to an order of magnitude.

Piccolo and Spark are distributed processing platforms that use table-/set-based abstractions, support it-

[21] `http://www.few.vu.nl/~e.krepska/HipG/`

[22] Parallel Boost Graph Library: `http://osl.iu.edu/research/pbgl/`

[23] `http://pargraph.sourceforge.net/`

[24] `http://www.boost.org/doc/libs/1_46_1/libs/graph/doc/index.html`

eration and can serve as the foundation of more specialized graph processing frameworks [44,55]. One such extension is the aforementioned Bagel which is built on Spark.

Also noteworthy are distributed data-flow engines such as Sawzall [43], Dryad/DryadLINQ [24,25], Pig [41], and Ciel/Skywriting [39]. Computations on graphs require a custom mapping to the respective data-flow language model. Some of the languages allow to express iterated processing, but the underlying systems are not optimised for doing this efficiently on graph structured data.

There are more graph processing libraries that focus on specific algorithms, but did not offer a detailed enough explanation of a more general programming model. Also we did not cover frameworks that focus on specific aspects of scaling algorithms on architectures such as supercomputers or GPUs, because the scalability challenges are different.

7. Limitations and Future Work

As we have seen in previous sections, SIGNAL/COLLECT is a scalable programming model and framework for parallel and distributed graph algorithms. Whilst our expressiveness evaluation is limited by the number of algorithms shown, their wide variability indicates some generalizability of our claim of simplicity and expressiveness. The generalizability of our scalability evaluations is also limited by the number of algorithms tested. But again, we believe that the range of algorithms is typical for such evaluations and provides a strong indication for SIGNAL/COLLECT's scalability.

In addition, our evaluation does raise some very interesting questions: Could we improve performance with a better graph partitioning scheme? How does SIGNAL/COLLECT fare with graphs containing vertices with disproportional numbers of in- and out-degree vertices? How could Signal/Collect recover from failures? And, what would the impact of a prioritising scheduler be on run-time performance? We discuss these questions in the remainder of this section.

Due to the default partitioning scheme the vast majority of signals in the distributed version are sent over the network. This is inefficient, but it could be improved without modifying the programming model: A domain optimised hash function could be used, for example one that maps websites from the same domain to the same worker or cluster node. This should improve locality of signaling. It would be interesting to see to what degree such a scheme would suffer from imbalanced loads for different domains.

We did not encounter any problems due to high in/out-degree vertices so far. If this should be the case at some point, then there would be different avenues to address this: One could solve the problem of *high in-degree vertices* by using Pregel-like combiners [38]. Such an approach should reduce the messaging and improve the performance of the distributed version even further. Such combiners are already supported by the framework, but they are at odds with the modularity of SIGNAL/COLLECT, where every vertex might have a different way to combine signals, which is not generally known to the worker from which the signals are sent. The problem of *high out-degree vertices* could be addressed by modifying a graph: High-out-degree vertices could create child vertices that each inherit a share of the outgoing edges. All state changes and further edge additions/removals are forwarded to the child vertices. High out-degree child vertices could recursively use the same scheme. A more efficient and more limited alternative is to parallelise the signaling on a vertex to "smear" the signaling workload across a cluster node instead of having the entire load on one worker.

SIGNAL/COLLECT currently only supports very primitive checkpointing and no error recovery. All the jobs that we have run so far were very short-lived and we never encountered hardware failures. Hence, for us it would make more sense to simply restart a failed job. With the long-running jobs and use cases such as query processing error recovery would become more important. The distributed snapshot algorithm [6] would probably be a good candidate for addressing this issue, because it could run without interrupting algorithm execution.

Finally, it would be interesting to experiment with a prioritising scheduler. Such a scheduler might have benefits for use cases in which computing and sending signals is very expensive relative to other tasks. Otherwise, the overhead of prioritising operations may not pay off.

8. Conclusions

Both researchers and industry are confronted with the need to process increasingly large amounts of data, much of which has a natural graph representation. In order to address the need to run algorithms on in-

creasingly large graphs we have designed a programming model that is both simple and expressive. We showed its expressiveness by designing adaptations of more than ten important algorithms to the programming model and the simplicity by being able to express these algorithms with just a few lines of code.

We built an open source framework that can parallelise and distribute the execution of algorithms formulated in the model. We empirically evaluated the scalability of the framework across different graph structures and algorithms and have shown that the framework scales with additional resources. The framework offers great efficiency and performance on a cluster architecture, which was shown by loading the huge Yahoo! AltaVista webgraph and computing high-quality PageRanks for its vertices in just 3 minutes on a dozen machines.

With SIGNAL/COLLECT we have created a programming abstraction that allows programmers to run algorithms quickly on large graphs without worrying about the specifics of how parallel and distributed processing resources are allocated. We believe that marrying actors with a graph abstraction should be taken beyond simple graph processing and that this is an effective approach to building dynamic and complex systems that operate on large data sets.

Acknowledgements

We would like to thank the Hasler foundation for funding our research and Yahoo! for giving us access to the AltaVista Web Page Hyperlink Connectivity Graph. We would like to thank Mihaela Verman, Lorenz Fischer, and Patrick Minder for the many interesting discussions, for feedback, and proofreading, as well as William Cohen for input on earlier versions of the project. We would also like to thank Francisco de Freitas for designing the first prototype of an Akka-based distributed version of the SIGNAL/COLLECT framework.

Appendix

In order to show how the framework is used in practice, we show the source code of an executable algorithm in the SIGNAL/COLLECT framework in Figure 22.[25]

```
import com.signalcollect._

class PageRankVertex(
  id: Any, initialState: Double)
  extends DataGraphVertex(id, initialState) {
  type Signal = Double
  def collect = 0.15 + 0.85 * signals.sum
}

class PageRankEdge(targetId: Any)
  extends DefaultEdge(targetId) {
  type Source = PageRankVertex
  def signal = source.state / source.edgeCount
}

object PageRankExample extends App {
  val graph = GraphBuilder.
    withStorageFactory(
      factory.storage.JavaMapStorage).
      build
  graph.addVertex(new PageRankVertex(1, 0.15))
  graph.addVertex(new PageRankVertex(2, 0.15))
  graph.addVertex(new PageRankVertex(3, 0.15))
  graph.addEdge(1, new PageRankEdge(2))
  graph.addEdge(2, new PageRankEdge(1))
  graph.addEdge(2, new PageRankEdge(3))
  graph.addEdge(3, new PageRankEdge(2))
  graph.execute(
    ExecutionConfiguration.
      withSignalThreshold(0.001))
  val top2 = graph.aggregate(
    TopKFinder[Double](k = 2))
  top2.foreach(println(_))
  graph.shutdown
}
```

Fig. 22. Executable Scala code of a PageRank algorithm definition, sequential graph building, a local execution, and an aggregation operation. Note that the actual PageRank code only encompasses the upper two class definitions. The `PageRankExample` object builds a tiny graph, executes the computation, runs an aggregation to find the two top ranked vertices, prints them, and then shuts the system down. Both the graph building and the execution are highly configurable: In this example an alternative storage implementation is used and the signal threshold is modified, both for illustration. The implementation could be simplified by using the defaults.

References

[1] ANDREEV, K. AND RÄCKE, H. 2004. Balanced graph partitioning. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*. SPAA '04. ACM, New York, NY, USA, 120–124.

[2] BIEMANN, C. 2006. Chinese whispers: an efficient graph clustering algorithm and its application to natural language process-

[25] Source code available at `https://github.com/uzh/signal-collect-evaluation/blob/master/src/main/scala/com/signalcollect/evaluation/algorithms/paper/PageRank.scala`.

ing problems. In *Proceedings of the First Workshop on Graph Based Methods for Natural Language Processing*. TextGraphs-1. Association for Computational Linguistics, Stroudsburg, PA, USA, 73–80.

[3] BU, Y., HOWE, B., BALAZINSKA, M., AND ERNST, M. D. 2010. Haloop: efficient iterative data processing on large clusters. *Proc. VLDB Endow. 3*, 285–296.

[4] CAI, Z., LOGOTHETIS, D., AND SIGANOS, G. 2012. Facilitating real-time graph mining. In *Proceedings of the fourth international workshop on Cloud data management*. CloudDB '12. ACM, New York, NY, USA, 1–8.

[5] CHAKRABARTI, S., DOM, B., AND INDYK, P. 1998. Enhanced hypertext categorization using hyperlinks. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*. SIGMOD '98. ACM, New York, NY, USA, 307–318.

[6] CHANDY, K. M. AND LAMPORT, L. 1985. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst. 3,* 1, 63–75.

[7] CHEN, R., WENG, X., HE, B., AND YANG, M. 2010. Large graph processing in the cloud. In *SIGMOD Conference'10*. 1123–1126.

[8] COHEN, J. 2009. Graph twiddling in a mapreduce world. *Computing in Science Engineering 11,* 4, 29 –41.

[9] DEAN, J. AND GHEMAWAT, S. 2004. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. USENIX Association, Berkeley, CA, USA, 10–10.

[10] DELORIMIER, M. 2013. Graph parallel actor language âĂŤ a programming language for parallel graph algorithms. Ph.D. thesis, California Institute of Technology. http://resolver.caltech.edu/CaltechTHESIS:08192012-145253489.

[11] DELORIMIER, M., KAPRE, N., MEHTA, N., RIZZO, D., ESLICK, I., RUBIN, R., URIBE, T. E., KNIGHT, T. F., AND DEHON, A. 2006. Graphstep: A system architecture for sparse-graph algorithms. In *In Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines. IEEE*. IEEE Computer Society.

[12] EKANAYAKE, J., LI, H., ZHANG, B., GUNARATHNE, T., BAE, S.-H., QIU, J., AND FOX, G. 2010. Twister: a runtime for iterative mapreduce. In *HPDC*, S. Hariri and K. Keahey, Eds. ACM, 810–818.

[13] GARDNER, M. 1970. Mathematical Games: The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 120–123.

[14] GETOOR, L., SEGAL, E., TASKAR, B., AND KOLLER, D. 2001. Probabilistic models of text and link structure for hypertext classification. In *In IJCAI Workshop on Text Learning: Beyond Supervision*.

[15] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. 2012. Powergraph: distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*. OSDI'12. USENIX Association, Berkeley, CA, USA, 17–30.

[16] GRANOVETTER, M. 1978. Threshold Models of Collective Behavior. *American Journal of Sociology 83,* 6, 1420–1443.

[17] GREGOR, D. AND LUMSDAINE, A. 2005a. Lifting sequential graph algorithms for distributed-memory parallel computation. *SIGPLAN Not. 40*, 423–437.

[18] GREGOR, D. AND LUMSDAINE, A. 2005b. The parallel bgl: A generic library for distributed graph computations. In *In Parallel Object-Oriented Scientific Computing (POOSC)*.

[19] GU, Y., LU, L., GROSSMAN, R., AND YOO, A. 2010. Processing massive sized graphs using sector/sphere. In *Many-Task Computing on Grids and Supercomputers (MTAGS), 2010 IEEE Workshop on*. 1 –10.

[20] HALLER, P. AND MILLER, H. 2011. Parallelizing machine learning– functionally: A framework and abstractions for parallel graph processing.

[21] HEWITT, C., BISHOP, P., AND STEIGER, R. 1973. A universal modular actor formalism for artificial intelligence. In *IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 235–245.

[22] HILBERT, M. AND LÓPEZ, P. 2011. The world's technological capacity to store, communicate, and compute information. *Science 332,* 6025, 60–65.

[23] HONG, S., CHAFI, H., SEDLAR, E., AND OLUKOTUN, K. 2012. Green-marl: a dsl for easy and efficient graph analysis. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVII. ACM, New York, NY, USA, 349–362.

[24] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. ACM, New York, NY, USA, 59–72.

[25] ISARD, M. AND YU, Y. 2009. Distributed data-parallel computing using a high-level programming language. In *Proceedings of the 35th SIGMOD international conference on Management of data*. SIGMOD '09. ACM, New York, NY, USA, 987–994.

[26] KAJDANOWICZ, T., INDYK, W., KAZIENKO, P., AND KUKUL, J. 2012. Comparison of the efficiency of mapreduce and bulk synchronous parallel approaches to large network processing. In *Data Mining Workshops (ICDMW), 2012 IEEE 12th International Conference on*. 218 –225.

[27] KANG, U., CHAU, D., AND FALOUTSOS, C. 2010. Inference of beliefs on billion-scale graphs. *The 2nd Workshop on Large-scale Data Mining: Theory and Applications*.

[28] KANG, U., TSOURAKAKIS, C. E., AND FALOUTSOS, C. 2009. Pegasus: A peta-scale graph mining system. *Data Mining, IEEE International Conference on 0*, 229–238.

[29] KOLLER, D. AND FRIEDMAN, N. 2009. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press.

[30] KREPSKA, E., KIELMANN, T., FOKKINK, W., AND BAL, H. 2011a. A high-level framework for distributed processing of large-scale graphs. In *Proceedings of the 12th international conference on Distributed computing and networking*. ICDCN'11. Springer-Verlag, Berlin, Heidelberg, 155–166.

[31] KREPSKA, E., KIELMANN, T., FOKKINK, W., AND BAL, H. 2011b. Hipg: parallel processing of large-scale graphs. *SIGOPS Oper. Syst. Rev. 45,* 2, 3–13.

[32] LIN, J. AND SCHATZ, M. 2010. Design patterns for efficient graph algorithms in mapreduce. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*. MLG '10. ACM, New York, NY, USA, 78–85.

[33] LOW, Y., GONZALEZ, J., KYROLA, A., BICKSON, D., AND GUESTRIN, C. 2011. Graphlab: A distributed framework for machine learning in the cloud. *CoRR abs/1107.0922*.

[34] LOW, Y., GONZALEZ, J., KYROLA, A., BICKSON, D.,

GUESTRIN, C., AND HELLERSTEIN, J. M. 2010. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*. Catalina Island, California.

[35] LUMSDAINE, A., GREGOR, D., HENDRICKSON, B., AND BERRY, J. W. 2007. Challenges in parallel graph processing. *Parallel Processing Letters*, 5–20.

[36] MACSKASSY, S. A. AND PROVOST, F. 2003. A simple relational classifier. In *Proceedings of the Second Workshop on Multi-Relational Data Mining (MRDM-2003) at KDD-2003*. 64–76.

[37] MACSKASSY, S. A. AND PROVOST, F. 2007. Classification in networked data: A toolkit and a univariate case study. *J. Mach. Learn. Res. 8*, 935–983.

[38] MALEWICZ, G., AUSTERN, M. H., BIK, A. J. C., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, A. K. Elmagarmid and D. Agrawal, Eds. ACM, 135–146.

[39] MURRAY, D. G., SCHWARZKOPF, M., SMOWTON, C., SMITH, S., MADHAVAPEDDY, A., AND HAND, S. 2011. Ciel: a universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*. NSDI'11. USENIX Association, Berkeley, CA, USA, 9–9.

[40] NAJORK, M., FETTERLY, D., HALVERSON, A., KENTHAPADI, K., AND GOLLAPUDI, S. 2012. Of hammers and nails: an empirical comparison of three paradigms for processing large graphs. In *Proceedings of the fifth ACM international conference on Web search and data mining*. WSDM '12. ACM, New York, NY, USA, 103–112.

[41] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. 2008. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. SIGMOD '08. ACM, New York, NY, USA, 1099–1110.

[42] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. 1998. The PageRank citation ranking: Bringing order to the Web. Tech. rep., Stanford Digital Library Technologies Project.

[43] PIKE, R., DORWARD, S., GRIESEMER, R., AND QUINLAN, S. 2005. Interpreting the data: Parallel analysis with sawzall. *Sci. Program. 13,* 4, 277–298.

[44] POWER, R. AND LI, J. 2010. Piccolo: building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. OSDI'10. USENIX Association, Berkeley, CA, USA, 1–14.

[45] RUSSELL, S. J. AND NORVIG, P. 2003. *Artificial Intelligence: a modern approach* 2nd international edition Ed. Prentice Hall.

[46] SALIHOGLU, S. AND WIDOM, J. 2012. Gps: A graph processing system. Tech. rep., Stanford, http://infolab.stanford.edu/gps/publications/tech-report.pdf.

[47] SCHURGAST, S. 2010. Markov logic inference on signal/collect. M.S. thesis, University of Zurich, Department of Informatics.

[48] SEO, S., YOON, E. J., KIM, J., JIN, S., KIM, J.-S., AND MAENG, S. 2010. Hama: An efficient matrix computation with the mapreduce framework. *Cloud Computing Technology and Science, IEEE International Conference on 0*, 721–726.

[49] SHAO, B., WANG, H., AND LI, Y. 2012. The trinity graph engine. Tech. Rep. 161291, Microsoft Research Asia.

[50] STREBEL, D. 2011. Making signal/collect scale.

[51] STUTZ, P., BERNSTEIN, A., AND COHEN, W. W. 2010. Signal/Collect: Graph Algorithms for the (Semantic) Web. In *International Semantic Web Conference (ISWC) 2010*, P. P.-S. et al., Ed. Vol. LNCS 6496. Springer, Heidelberg, pp. 764–780.

[52] STUTZ, P., VERMAN, M., FISCHER, L., AND BERNSTEIN, A. Triplerush: A fast and scalable triple store. In *9th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2013)*. 50.

[53] VALIANT, L. G. 1990. A bridging model for parallel computation. *Commun. ACM 33,* 8, 103–111.

[54] WANG, G., XIE, W., DEMERS, A., AND GEHRKE, J. 2013. Asynchronous large-scale graph processing made easy. In *CIDR*.

[55] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. 2010. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. HotCloud'10. USENIX Association, Berkeley, CA, USA, 10–10.

[56] ZHANG, Y., GAO, Q., GAO, L., AND WANG, C. 2011a. imapreduce: A distributed computing framework for iterative computation. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*. IPDPSW '11. IEEE Computer Society, Washington, DC, USA, 1112–1121.

[57] ZHANG, Y., GAO, Q., GAO, L., AND WANG, C. 2011b. Priter: a distributed framework for prioritized iterative computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. SOCC '11. ACM, New York, NY, USA, 13:1–13:14.

[58] ZHU, X. AND GHAHRAMANI, Z. 2002. Learning from labeled and unlabeled data with label propagation. Tech. rep.