Distributed REScala: un update algorithm for distributed Reactive Programming

(José Proença @ 1 April)

OOPSLA’14 - Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini
(TU-Darmstad)
Context and Motivation

Lack of generic distributed algorithms
- only client-side, no glitch-freedom, or cannot be distributed

Distributed RP vs. observers + remote obj.
Contributions

Algorithms for distributed glitch-freedom:

- **Scala.React**
  - Topological sorting + priority queue

- **Scala.Rx**
  - Parallel propagator version

- **ELM**
  - Decentralised flooding

- **SID-UP**
  - NEW: send & store more data

Evaluation
Dynamic dependencies

\[ z = \text{if } c \text{ then } x \text{ else } y \]

- \( c \) is a boolean value that can change dynamically, whenever \( c \) changes, \( x \) or \( y \) can be updated.
- \( c \) can be updated from different sources, making it hard to maintain consistency.
- \( c \) is set to true or false, affecting the evaluation of \( x \) or \( y \).

Conditionals

High-order reactives

getPerson()
- .getNames()
- .selectFst()
Scala.React

Topological sorting + priority queue

Figure 7. Visual comparison of update propagation with different algorithms.

4.1 Example-Based Comparison

To highlight the advantages of SID-UP, Figure 7 shows how an update propagates through a DG with different approaches. Bold edges represent pulse notifications pending processing by their receiving node. Bold-outlined nodes have just processed their incoming notifications and pulsed. Nodes shaded gray have reevaluated.

Scala.React and Scala.Rx (1st and 2nd timeline) proceed in topological order. The processing layer is highlighted by the overlapping rectangle. Scala.React is single-threaded: Only a single node in the rectangle is updating (shown in bold) at any point in time. In Scala.Rx all nodes on the same layer are updated concurrently, reducing the number of steps the algorithm requires in trade for some synchronization overhead after each level. In Scala.React and Scala.Rx bold dependency edges and bold-outlined nodes correspond to messages that have to be transmitted: The former are transmitted between the nodes at each end of the edge, the latter between the node and the centralized priority queue.

As outlined at the end of Section 2.4, ELM in its original form is not suitable for the distributed setting, because its pipelining feature renders HORs impossible, and is actually incomparable to the other algorithms. Yet, for completeness we include it in the comparison, but make sure that its differences to the other algorithms show no effect by looking at a single turn without topology changes in the graph. ELM (3rd line) does not have a priority queue that restricts reevaluations to a single active layer. As can be seen from the figure, nodes update in different layers of the graph. As nodes reevaluate after each incoming edge is bold without authorization from a coordinator, only bold dependency edges correspond to messages. Nodes with a bold outline do
Scala.React

Topological sorting + priority queue

Figure 7. Visual comparison of update propagation with different algorithms.

4.1 Example-Based Comparison
To highlight the advantages of SID-UP, Figure 7 shows how an update propagates through a DG with different approaches. Bold edges represent pulse notifications pending processing by their receiving node. Bold-outlined nodes have just processed their incoming notifications and pulsed. Nodes shaded gray have reevaluated. Scala.React and Scala.Rx (1st and 2nd timeline) proceed in topological order. The processing layer is highlighted by the overlapping rectangle. Scala.React is single-threaded: Only a single node in the rectangle is updating (shown in bold) at any point in time. In Scala.Rx all nodes on the same layer are updated concurrently, reducing the number of steps the algorithm requires in trade for some synchronization overhead after each level. In Scala.React and Scala.Rx bold dependency edges and bold-outlined nodes correspond to messages that have to be transmitted: The former are transmitted between the nodes at each end of the edge, the latter between the node and the centralized priority queue.

As outlined at the end of Section 2.4, ELM in its original form is not suitable for the distributed setting, because its pipelining feature renders HORs impossible, and is actually incomparable to the other algorithms. Yet, for completeness we include it in the comparison, but make sure that its differences to the other algorithms show no effect by looking at a single turn without topology changes in the graph. ELM (3rd line) does not have a priority queue that restricts reevaluations to a single active layer. As can be seen from the figure, nodes update in different layers of the graph. As nodes reevaluate after each incoming edge is bold without authorization from a coordinator, only bold dependency edges correspond to messages. Nodes with a bold outline do not have a priority queue that restricts reevaluations to a single active layer. As can be seen from the figure, nodes update in different layers of the graph. As nodes reevaluate after each incoming edge is bold without authorization from a coordinator, only bold dependency edges correspond to messages. Nodes with a bold outline do not have a priority queue that restricts reevaluations to a single active layer.
4.1 Example-Based Comparison

To highlight the advantages of SID-UP, Figure 7 shows how an update propagates through a DG with different approaches. Bold edges represent pulse notifications pending processing by their receiving node. Bold-outlined nodes have just processed their incoming notifications and pulsed. Nodes shaded gray have reevaluated.

Scala.React and Scala.Rx (1st and 2nd timeline) proceed in topological order. The processing layer is highlighted by the overlapping rectangle. Scala.React is single-threaded: Only a single node in the rectangle is updating (shown in bold) at any point in time. In Scala.Rx all nodes on the same layer are updated concurrently, reducing the number of steps the algorithm requires in trade for some synchronization overhead after each level. In Scala.React and Scala.Rx bold dependency edges and bold-outlined nodes correspond to messages that have to be transmitted: The former are transmitted between the nodes at each end of the edge, the latter between the node and the centralized priority queue.

As outlined at the end of Section 2.4, ELM in its original form is not suitable for the distributed setting, because its pipelining feature renders HORs impossible, and is actually incomparable to the other algorithms. Yet, for completeness we include it in the comparison, but make sure that its differences to the other algorithms show no effect by looking at a single turn without topology changes in the graph. ELM (3rd line) does not have a priority queue that restricts reevaluations to a single active layer. As can be seen from the figure, nodes update in different layers of the graph. As nodes reevaluate after each incoming edge is bold without authorization from a coordinator, only bold dependency edges correspond to messages. Nodes with a bold outline do not have a priority queue and therefore reevaluate after each incoming edge in the graph.
Scala.React

Topological sorting + priority queue

Figure 7. Visual comparison of update propagation with different algorithms.

4.1 Example-Based Comparison

To highlight the advantages of SID-UP, Figure 7 shows how an update propagates through a DG with different approaches. Bold edges represent pulse notifications pending processing by their receiving node. Bold-outlined nodes have just processed their incoming notifications and pulsed. Nodes shaded gray have reevaluated.

Scala.React and Scala.Rx (1st and 2nd timeline) proceed in topological order. The processing layer is highlighted by the overlapping rectangle. Scala.React is single-threaded: Only a single node in the rectangle is updating (shown in bold) at any point in time. In Scala.Rx all nodes on the same layer are updated concurrently, reducing the number of steps the algorithm requires in trade for some synchronization overhead after each level. In Scala.React and Scala.Rx bold dependency edges and bold-outlined nodes correspond to messages that have to be transmitted: The former are transmitted between the nodes at each end of the edge, the latter between the node and the centralized priority queue. As outlined at the end of Section 2.4, ELM in its original form is not suitable for the distributed setting, because its pipelining feature renders FORs impossible, and is actually incomparable to the other algorithms. Yet, for completeness we include it in the comparison, but make sure that its differences to the other algorithms show no effect by looking at a single turn without topology changes in the graph. ELM (3rd line) does not have a priority queue that restricts reevaluations to a single active layer. As can be seen from the figure, nodes update in different layers of the graph. As nodes reevaluate after each incoming edge is bold without authorization from a coordinator, only bold dependency edges correspond to messages. Nodes with a bold outline do
Scala.Rx
Parallel propagator version

Figure 7. Visual comparison of update propagation with different algorithms. Bold edges represent pulse notifications pending processing by their receiving node. Bold-outlined nodes have just processed their incoming notifications and pulsed. Nodes shaded gray have reevaluated.

Scala.React and Scala.Rx (1st and 2nd timeline) proceed in topological order. The processing layer is highlighted by the overlapping rectangle. Scala.React is single-threaded: Only a single node in the rectangle is updating (shown in bold) at any point in time. In Scala.Rx all nodes on the same layer are updated concurrently, reducing the number of steps the algorithm requires in trade for some synchronization overhead after each level. In Scala.React and Scala.Rx bold dependency edges and bold-outlined nodes correspond to messages that have to be transmitted: The former are transmitted between the nodes at each end of the edge, the latter between the node and the centralized priority queue.

As outlined at the end of Section 2.4, ELM in its original form is not suitable for the distributed setting, because its pipelining feature renders HORs impossible, and is actually incomparable to the other algorithms. Yet, for completeness we include it in the comparison, but make sure that its differences to the other algorithms show no effect by looking at a single turn without topology changes in the graph. ELM (3rd line) does not have a priority queue that restricts reevaluations to a single active layer. As can be seen from the figure, nodes update in different layers of the graph. As nodes reevaluate after each incoming edge is bold without authorization from a coordinator, only bold dependency edges correspond to messages. Nodes with a bold outline do entire layer at a time!
Scala.Rx

Parallel propagator version

Scala.Rx: process layer-wise, entire layer at a time. 8 steps. 28 messages.
4.1 Example-Based Comparison

To highlight the advantages of SID-UP, Figure 7 shows how an update propagates through a DG with different approaches. Bold edges represent pulse notifications pending processing by their receiving node. Bold-outlined nodes have just processed their incoming notifications and pulsed. Nodes shaded gray have reevaluated. Scala.React and Scala.Rx (1st and 2nd timeline) proceed in topological order. The processing layer is highlighted by the overlapping rectangle. Scala.React is single-threaded: Only a single node in the rectangle is updating (shown in bold) at any point in time. In Scala.Rx all nodes on the same layer are updated concurrently, reducing the number of steps the algorithm requires in trade for some synchronization overhead after each level. In Scala.React and Scala.Rx bold dependency edges and bold-outlined nodes correspond to messages that have to be transmitted: The former are transmitted between the nodes at each end of the edge, the latter between the node and the centralized priority queue.

As outlined at the end of Section 2.4, ELM in its original form is not suitable for the distributed setting, because its pipelining feature renders HORs impossible, and is actually incomparable to the other algorithms. Yet, for completeness we include it in the comparison, but make sure that its differences to the other algorithms show no effect by looking at a single turn without topology changes in the graph. ELM (3rd line) does not have a priority queue that restricts reevaluations to a single active layer. As can be seen from the figure, nodes update in different layers of the graph. As nodes reevaluate after each incoming edge is bold without authorization from a coordinator, only bold dependency edges correspond to messages. Nodes with a bold outline do not have a priority queue and reevaluate independently.

Round initiator

Each node sends:
- “change” pulse, or
- “no-change” pulse
4.1 Example-Based Comparison

To highlight the advantages of SID-UP, Figure 7 shows how an update propagates through a DG with different approaches. Bold edges represent pulse notifications pending processing by their receiving node. Bold-outlined nodes have just processed their incoming notifications and pulsed. Nodes shaded gray have reevaluated.

Scala.React and Scala.Rx (1st and 2nd timeline) proceed in topological order. The processing layer is highlighted by the overlapping rectangle. Scala.React is single-threaded: Only a single node in the rectangle is updating (shown in bold) at any point in time. In Scala.Rx all nodes on the same layer are updated concurrently, reducing the number of steps the algorithm requires in trade for some synchronization overhead after each level. In Scala.React and Scala.Rx bold dependency edges and bold-outlined nodes correspond to messages that have to be transmitted: The former are transmitted between the nodes at each end of the edge, the latter between the node and the centralized priority queue.

As outlined at the end of Section 2.4, ELM in its original form is not suitable for the distributed setting, because its pipelining feature renders HORs impossible, and is actually incomparable to the other algorithms. Yet, for completeness we include it in the comparison, but make sure that its differences to the other algorithms show no effect by looking at a single turn without topology changes in the graph. ELM (3rd line) does not have a priority queue that restricts reevaluations to a single active layer. As can be seen from the figure, nodes update in different layers of the graph. As nodes reevaluate after each incoming edge is bold without authorization from a coordinator, only bold dependency edges correspond to messages. Nodes with a bold outline do not wait for message from every incoming edge. 7 steps. 37 messages.
ELM
Decentralised flooding

Supports pipelining  BUT  No high-order reactives

\( ELM^S \)
Supports HOR  BUT  No pipelining

Round initiator
blocks until the end of the round
4.1 Example-Based Comparison

To highlight the advantages of SID-UP, Figure 7 shows how an update propagates through a DG with different approaches. Bold edges represent pulse notifications pending processing by their receiving node. Bold-outlined nodes have just processed their incoming notifications and pulsed. Nodes shaded gray have reevaluated.

Scala.React and Scala.Rx (1st and 2nd timeline) proceed in topological order. The processing layer is highlighted by the overlapping rectangle. Scala.React is single-threaded: Only a single node in the rectangle is updating (shown in bold) at any point in time. In Scala.Rx all nodes on the same layer are updated concurrently, reducing the number of steps the algorithm requires in trade for some synchronization overhead after each level. In Scala.React and Scala.Rx bold dependency edges and bold-outlined nodes correspond to messages that have to be transmitted: The former are transmitted between the nodes at each end of the edge, the latter between the node and the centralized priority queue.

As outlined at the end of Section 2.4, ELM in its original form is not suitable for the distributed setting, because its pipelining feature renders HORs impossible, and is actually incomparable to the other algorithms. Yet, for completeness we include it in the comparison, but make sure that its differences to the other algorithms show no effect by looking at a single turn without topology changes in the graph. ELM (3rd line) does not have a priority queue that restricts reevaluations to a single active layer. As can be seen from the figure, nodes update in different layers of the graph. As nodes reevaluate after each incoming edge is bold without authorization from a coordinator, only bold dependency edges correspond to messages. Nodes with a bold outline do...
SID-UP

Node states:
- pending (not pulsed),
- changed, or unchanged

When evaluated, each node sends:
- changed + changed sources, or
- unchanged + changed sources
The novelty of SID-UP consists of the combination of the following properties:

- Limited amount of remote communication – less than one-to-one communication.
- Freedom) for node re-evaluations
- Exploitation of concurrency potential (respecting glitch-freedom both in local and distributed settings
- Support for distributed reactive programming with re-actives for dynamic network topologies
- No unbounded one-to-many communication
- Glitch-freedom both in local and distributed settings
- Support for fully-fledged reactive programming, including completely unaffected ones, is involved in every change at a time, ELM supports.
- No centralized coordinator during propagation phases

The design of SID-UP was driven by two goals: (a) Reduce the amount of remote communication completely. The algorithm still relies on unbounded one-to-many communication and still enforces a lot of wasted computational resources, as messages have to be processed simultaneously every update turn at every node implies a lot of wasted

In ELM [9], a central coordinator broadcasts every update turn's start to all input nodes. After the admission phase, all changed input nodes send out a "change" pulse and all unchanged input nodes do not have a path of dependencies between them should not need to wait for each other. This feature is especially desirable in the distributed set-

In SID-UP, the coordinator for the admission phase to be converted to a distributed version would require including completely unaffected ones, is involved in every propagation phase, meaning no unnecessary sequentiality: Nodes on different layers that only update when at least one of their incoming dependencies sent a pulse instead. Hence, every node, in-

During the propagation phase, the chain of calls. However, the algorithm still relies on un-

Figure 6. SID-UP - remote propagation algorithm.
SID-UP

- glitch-freedom
- No unbounded one-to-many communication
- No coordinator when propagating
- Faster than Scala.*
- As fast as ELM
- Less messages
- Support for dynamic dependencies
- No pipelining
- Finite & acyclic dep. graphs
Performance cost

The comparison against the unsafe observer implementation shows that the above advantages still apply to software that does not require glitch freedom. Removing the manual glitch freedom implementation lessens the gap in terms of code metrics. Yet, column unsafe observer still exhibits worse metrics than reactive.

Performance Cost. For the performance comparison, we inspected again only the part of the application concerning remote value propagation. Both the application responsiveness for the user as well as the amount of data throughput depend mainly on the time it takes for every input value change to be propagated through the entire application. To measure this time, we implemented a loop that publishes a new list of orders at the order depot whenever the previous update had taken effect in management (cf. Figure 1).

Figure 9 shows the amount of time required to perform an increasing number of subsequent update turns. Each update turn sets a new list of ten orders in the order depot. Unsurprisingly, the implementation based on remote observers is faster in completing pushing updates through the application. This is easily explained, as the generality of reactive programming comes at a cost. In our case, this cost consists of the operations performed with all the source identifier sets on each node. Though, as the graph shows, the processing time of reactive programming supported by SID-UP still exhibits similar complexity and only increases required time by a factor of approximately 10% in this case.

5.2 Benchmarks

To empirically compare SID-UP with existing algorithms, we implemented a benchmark in form of a reactive network through which we propagate an update. The algorithms being compared are expected to perform differently on different topologies due to various approaches for parallelism and message transmissions. Hence, we assembled a graph from modules implementing various topologies. Figure 10 shows a schema of the graph we used. The “chain” module implements a linear chain of reactive nodes that does not allow parallelism. The “regular” module implements a graph with some nodes connected with varying degrees of fan-in and fan-out dependency connections that allows some parallelism. To be realistic, these degrees are chosen according to statistic distributions, which we measured by instrumenting 20 local reactive applications we developed in previous case studies; Figure 11 shows the distribution of the number of nodes for each value of incoming degree (left) and outgoing degree (right). The “fan” module implements a topology where one node fans out into a lot of immediate successors, all of which can be reevaluated concurrently. Each module contains 25 nodes and both “regular” and “fan” contain a few nodes whose values do not change during the update turn, i.e. although a dependency changes, they update to an unchanged pulse value or equivalently do not add their outgoing dependencies to the priority queue. Updates can be initiated inside each module separately through individual source nodes, although for the duration benchmarks we always update all sources to affect the entire graph. Finally, a dependent node at the end unifies the updates from all modules to detect the update turn completion.

Optimizations of the dependency graph’s topology may improve the performance of reactive programs. Since typically each node in the graph introduces a certain amount of overhead, optimizing the topology by reducing the number of nodes leads to better performance. The most prominent technique here is Lowering [6]. We performed our analyses without applying such techniques, as they are orthogonal to the focus of this paper.

---

only ± 10% slower than observers + manual glitch control
Benchmarks

Figure 9. Performance of the case study.

The comparison against the unsafe observer implementation shows that the above advantages still apply to software that does not require glitch freedom. Removing the manual glitch freedom implementation lessens the gap in terms of code metrics. Yet, column unsafe observer still exhibits worse metrics than reactive.

Performance Cost. For the performance comparison, we inspected again only the part of the application concerning remote value propagation. Both the application responsiveness for the user as well as the amount of data throughput depend mainly on the time it takes for every input value change to be propagated through the entire application. To measure this time, we implemented a loop that publishes a new list of orders at the order depot whenever the previous update had taken effect in management (cf. Figure 1).

Figure 9 shows the amount of time required to perform an increasing number of subsequent update turns. Each update turn sets a new list of ten orders in the order depot. Unsurprisingly, the implementation based on remote observers is faster in completing pushing updates through the application. This is easily explained, as the generality of reactive programming comes at a cost. In our case, this cost consists of the operations performed with all the source identifier sets on each node. Though, as the graph shows, the processing time of reactive programming supported by SID-UP still exhibits similar complexity and only increases required time by a factor of approximately 10% in this case.

5.2 Benchmarks

To empirically compare SID-UP with existing algorithms, we implemented a benchmark in form of a reactive network through which we propagate an update. The algorithms being compared are expected to perform differently on different topologies due to various approaches for parallelism and message transmissions. Hence, we assembled a graph from modules implementing various topologies. Figure 10 shows a schema of the graph we used. The "chain" module implements a linear chain of reactive nodes that does not allow low parallelism. The "regular" module implements a graph with some nodes connected with varying degrees of fan-in and fan-out dependency connections that allows some parallelism. To be realistic, these degrees are chosen according to statistic distributions, which we measured by instrumenting 20 local reactive applications we developed in previous case studies; Figure 11 shows the distribution of the number of nodes for each value of incoming degree (left) and outgoing degree (right). The "fan" module implements a topology where one node fans out into a lot of immediate successors, all of which can be reevaluated concurrently. Each module contains 25 nodes and both "regular" and "fan" contain a few nodes whose values do not change during the update turn, i.e. although a dependency changes, they update to an unchanged pulse value or equivalently do not add their outgoing dependencies to the priority queue. Updates can be initiated inside each module separately through individual source nodes, although for the duration benchmarks we always update all sources to affect the entire graph. Finally, a dependent node at the end unifies the updates from all modules to detect the update turn completion.

Optimizations of the dependency graph's topology may improve the performance of reactive programs. Since typically each node in the graph introduces a certain amount of overhead, optimizing the topology by reducing the number of nodes leads to better performance. The most prominent technique here is Lowering [6]. We performed our analyses without applying such techniques, as they are orthogonal.

Simulated network latency

\[3 + 25 \times 3 + 1\]

based on 20 previous case-studies

some nodes do not produce changes
which caused linearly increasing execution time.

To run the benchmarks, we fixed a bug of Scala.Rx's garbage collection support present in commit e4f4070cac cloned on 11/26/2013, RAM. Windows 7 64 bit and an Intel Core i5-3320M with 8 GB of 3

over the network, i.e. whenever a node sends a pulse notifi-

waiting times wherever a method call would have to be send

ports distribution, we simulate network latency by injecting

separate host. As no framework except

nodes as well as the coordinator (if needed) run on another

each module runs on an individual host; the source and sink

putational overhead per node.

the same turn durations in terms of user computations and

UP

sure the overhead caused by the set operations used in

UP

mentation of the parallel propagation strategy and our ref-

Scala.React's and Scala.Rx's implementation of sequential

optimizations and propagation algorithms would be an in-

interesting study, but extends beyond the scope of this paper.

involvement from the run-time propagation algorithm and

can be computed statically. That said, we expect such opti-

mizations to benefit

whether or not the graph has had some nodes

collapsed into fewer nodes does not change these interac-

mized to not. Whether or not the graph has had some nodes

equilibrium from the run-time propagation algorithm and opti-

mical to the propagation algorithm in use. Each algorithm

involves the graph in Figure 10 by pretending that

The experiments compare five propagation algorithms

: As we showed in Section 4.3,

ELM hybrid algorithm ELM

s

exhibit

SID-UP

and ELM

actually sup-

s

from Section 4.2 to mea-

to estimate the overhead caused by the more complex set op-

mote message delays. But, both

mark-and-sweep algorithm in the comparison, as executing

logical sorting [27]. We did not include a straight-forward

lowest factor.

scaling with latency, with

ferences between these curves indicate different factors of

network edges.

Figure 10 when simulating various amounts of delay on the

time it takes to complete an update turn on the graph in Fig-

happens outside of the dependency edges between nodes.

from remote communication with the priority queue, which

advantage of the topological sorting-based algorithms stems

edges are actually remote connections. But, the main dis-

This may seem like a disadvantageous comparison for algo-

sends a command to a node on a different host than its own.

The left graph in Figure 12 shows for each algorithm the

The right graph in Figure 12 shows how the time to complete a

update varies with the number of sources.

Table 1: Timex(ms) for different combinations of topology

<table>
<thead>
<tr>
<th>Topology</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SID-UP</td>
<td>10</td>
</tr>
<tr>
<td>scala.react</td>
<td>10</td>
</tr>
<tr>
<td>scala.rx</td>
<td>10</td>
</tr>
<tr>
<td>scala.rx parallel</td>
<td>10</td>
</tr>
<tr>
<td>ELMs</td>
<td>10</td>
</tr>
</tbody>
</table>
Benchmarks

**Time to complete 1 run**

- **3 + 25*3 + 1**
- **250 + 3 + 25*3 + 1**
To run the benchmarks, we fixed a bug of Scala.Rx's garbage collection support present in commit e4f4070cac cloned on 11/26/2013, as there is barely a difference, this comparison indicates that this overhead is negligible.

We also include our implementation of the parallel propagation strategy and our reference implementation of Scala.React's and Scala.Rx's implementation of sequential optimizations and propagation algorithms would be an interesting study, but extends beyond the scope of this paper.

The left graph in Figure 12 shows for each algorithm the same turn durations in terms of user computations and the most of all included algorithms, we also include our update. But, because of double logarithmic scaling, the differences between these curves indicate different factors of scaling with latency, with other algorithms, especially considering the double logarithmic scaling.

As there is barely a difference, this comparison indicates that this overhead is negligible. But, the advantage of the topological sorting-based algorithms stems from remote communication with the priority queue, which happens outside of the dependency edges between nodes.

This is in line with previous findings that mark-and-sweep algorithms generally outperform those based on topological sorting, although there are exceptions. Simulations are implicitly marked through the set intersection of the same turn durations in terms of user computations and the most of all included algorithms, we also include our update. But, because of double logarithmic scaling, the differences between these curves indicate different factors of scaling with latency, with other algorithms, especially considering the double logarithmic scaling.

The sweep phase is done implicitly: In mark-and-sweep algorithms, simply every node is implicitly considered as a special case of mark-and-sweep algorithms, where the sweep phase is done implicitly: In mark-and-sweep algorithm in the comparison, as executing logical sorting generally outperforms those based on topological sorting. We did not include a straight-forward sweep algorithm.

As there is barely a difference, this comparison indicates that this overhead is negligible. But, the advantage of the topological sorting-based algorithms stems from remote communication with the priority queue, which happens outside of the dependency edges between nodes.

This may seem like a disadvantageous comparison for algorithms dedicated to distributed graphs, since only very few algorithms can be computed statically. That said, we expect such optimizations to benefit collapsed into fewer nodes does not change these interactions. Whether or not the graph has had some nodes involving from the run-time propagation algorithm and optimizations do not require any involvement from the run-time propagation algorithm and optimizations.

The experiments compare five propagation algorithms: As we showed in Section 4.3, differences between these curves indicate different factors of scaling with latency, with other algorithms, especially considering the double logarithmic scaling.

There are special cases of mark-and-sweep algorithms, where the sweep phase is done implicitly: In mark-and-sweep algorithm in the comparison, as executing logical sorting generally outperforms those based on topological sorting. We did not include a straight-forward sweep algorithm.

As there is barely a difference, this comparison indicates that this overhead is negligible. But, the advantage of the topological sorting-based algorithms stems from remote communication with the priority queue, which happens outside of the dependency edges between nodes.

This may seem like a disadvantageous comparison for algorithms dedicated to distributed graphs, since only very few algorithms can be computed statically. That said, we expect such optimizations to benefit collapsed into fewer nodes does not change these interactions. Whether or not the graph has had some nodes involving from the run-time propagation algorithm and optimizations do not require any involvement from the run-time propagation algorithm and optimizations.

Whether or not the graph has had some nodes involving from the run-time propagation algorithm and optimizations do not require any involvement from the run-time propagation algorithm and optimizations.

As there is barely a difference, this comparison indicates that this overhead is negligible. But, the advantage of the topological sorting-based algorithms stems from remote communication with the priority queue, which happens outside of the dependency edges between nodes.

This may seem like a disadvantageous comparison for algorithms dedicated to distributed graphs, since only very few algorithms can be computed statically. That said, we expect such optimizations to benefit collapsed into fewer nodes does not change these interactions. Whether or not the graph has had some nodes involving from the run-time propagation algorithm and optimizations do not require any involvement from the run-time propagation algorithm and optimizations.

As there is barely a difference, this comparison indicates that this overhead is negligible. But, the advantage of the topological sorting-based algorithms stems from remote communication with the priority queue, which happens outside of the dependency edges between nodes.

This may seem like a disadvantageous comparison for algorithms dedicated to distributed graphs, since only very few algorithms can be computed statically. That said, we expect such optimizations to benefit collapsed into fewer nodes does not change these interactions. Whether or not the graph has had some nodes involving from the run-time propagation algorithm and optimizations do not require any involvement from the run-time propagation algorithm and optimizations.

As there is barely a difference, this comparison indicates that this overhead is negligible. But, the advantage of the topological sorting-based algorithms stems from remote communication with the priority queue, which happens outside of the dependency edges between nodes.

This may seem like a disadvantageous comparison for algorithms dedicated to distributed graphs, since only very few algorithms can be computed statically. That said, we expect such optimizations to benefit collapsed into fewer nodes does not change these interactions. Whether or not the graph has had some nodes involving from the run-time propagation algorithm and optimizations do not require any involvement from the run-time propagation algorithm and optimizations.

As there is barely a difference, this comparison indicates that this overhead is negligible. But, the advantage of the topological sorting-based algorithms stems from remote communication with the priority queue, which happens outside of the dependency edges between nodes.

This may seem like a disadvantageous comparison for algorithms dedicated to distributed graphs, since only very few algorithms can be computed statically. That said, we expect such optimizations to benefit collapsed into fewer nodes does not change these interactions. Whether or not the graph has had some nodes involving from the run-time propagation algorithm and optimizations do not require any involvement from the run-time propagation algorithm and optimizations.

As there is barely a difference, this comparison indicates that this overhead is negligible. But, the advantage of the topological sorting-based algorithms stems from remote communication with the priority queue, which happens outside of the dependency edges between nodes.

This may seem like a disadvantageous comparison for algorithms dedicated to distributed graphs, since only very few algorithms can be computed statically. That said, we expect such optimizations to benefit collapsed into fewer nodes does not change these interactions. Whether or not the graph has had some nodes involving from the run-time propagation algorithm and optimizations do not require any involvement from the run-time propagation algorithm and optimizations.

As there is barely a difference, this comparison indicates that this overhead is negligible. But, the advantage of the topological sorting-based algorithms stems from remote communication with the priority queue, which happens outside of the dependency edges between nodes.

This may seem like a disadvantageous comparison for algorithms dedicated to distributed graphs, since only very few algorithms can be computed statically. That said, we expect such optimizations to benefit collapsed into fewer nodes does not change these interactions. Whether or not the graph has had some nodes involving from the run-time propagation algorithm and optimizations do not require any involvement from the run-time propagation algorithm and optimizations.

As there is barely a difference, this comparison indicates that this overhead is negligible. But, the advantage of the topological sorting-based algorithms stems from remote communication with the priority queue, which happens outside of the dependency edges between nodes.
Benchmarks

Number of messages

to show it is better than ELMs
Some discussion

**ELM**: pipelining support - not evaluated (only 1 run)

**Scala.***: pipelining possible?

**ELMs + SID-UP**: How to detect the end of a run?

**Lots of changed sources**: not tested

**Scalability? (lock at each round)**

**Dynamic dependencies**: can affect topology? Assumptions?

**distributed clock?**
Optimisations

REBLS ’14 - Splash workshop

“Optimizing Distributed REScala”
Joscha Drechsler and Guido Salvaneschi

If has 1 incoming dep.:

- avoid iterations, intersections, waiting

If has no dyn. dep.:

- just count incoming pulses

include valueChanged and sourcesChanged in pulse
Optimisations

REBLS ’14 - Splash workshop

“Optimizing RESScala”
Joscha Drechsler and Guido Salvaneschi

If has 1 incoming dep.:
- avoid iterations, intersections, waiting

If has no dyn. dep.:
- just count incoming pulses

Benchmarks without injected network delays

include valueChanged and sourcesChanged in pulse