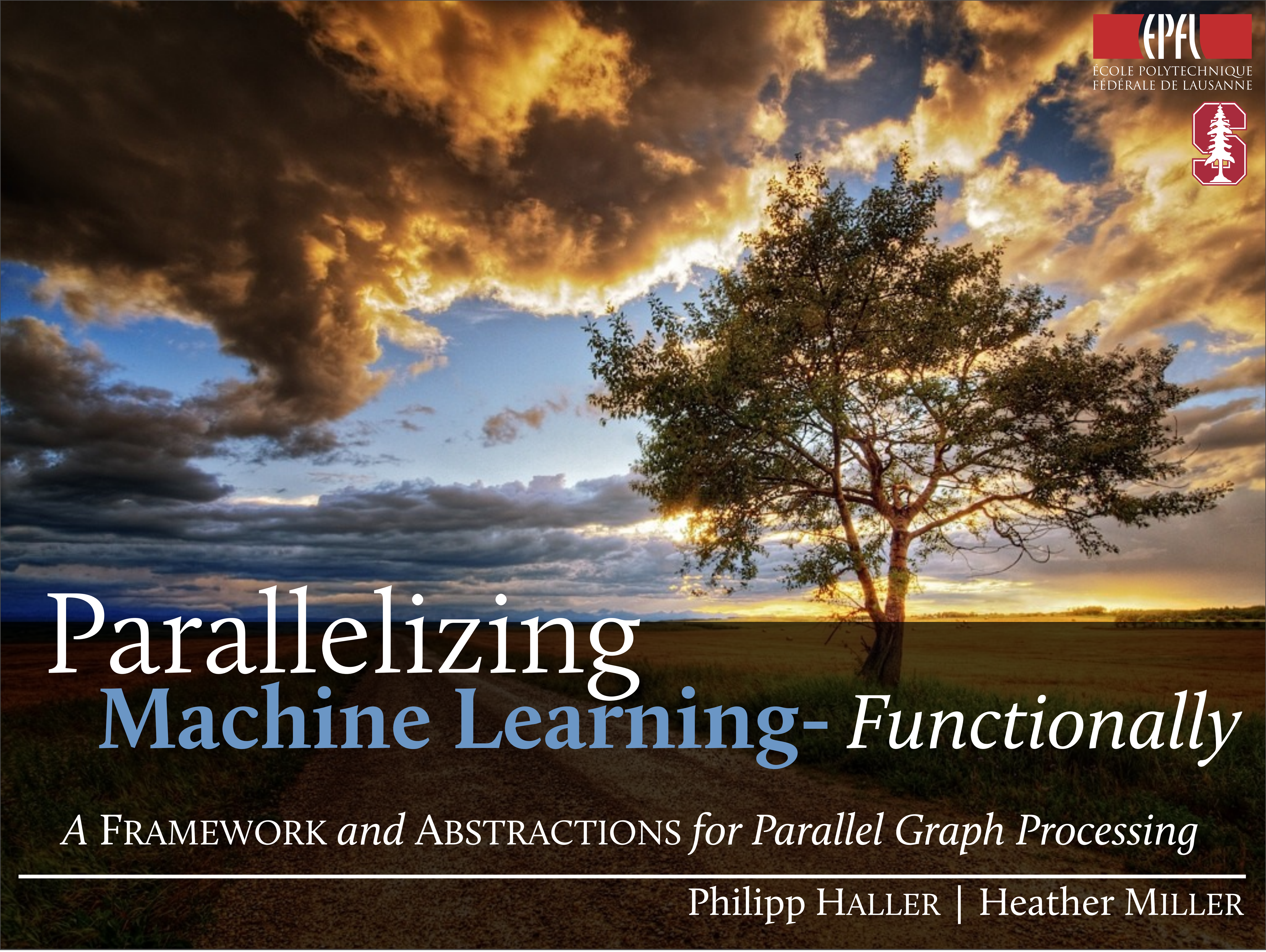


ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

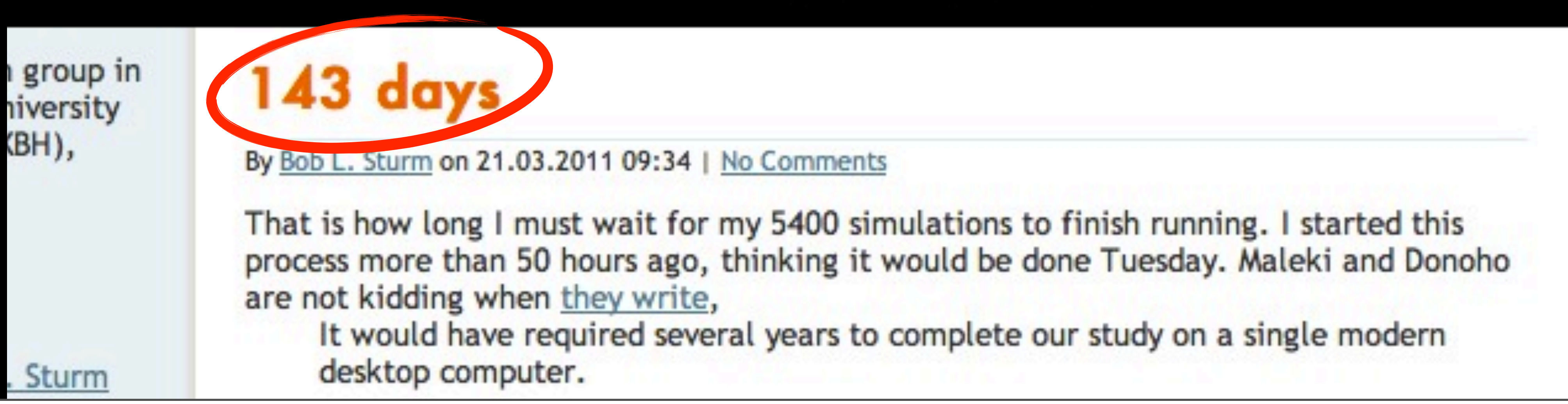
Parallelizing Machine Learning- *Functionally*

A FRAMEWORK *and* ABSTRACTIONS *for Parallel Graph Processing*

Philipp HALLER | Heather MILLER

Data is growing.

At the same time,
there is a growing desire
to ***do MORE with that data.***



143 days

By Bob L. Sturm on 21.03.2011 09:34 | No Comments

That is how long I must wait for my 5400 simulations to finish running. I started this process more than 50 hours ago, thinking it would be done Tuesday. Maleki and Donoho are not kidding when they write,

> It would have required several years to complete our study on a single modern desktop computer.

As an example,

Machine Learning (ML)

- has provided elegant and sophisticated solutions to many complex problems on a small scale,

As an example,

Machine Learning (ML)

- has provided elegant and sophisticated solutions to many complex problems on a small scale,

could open up New Applications + New Avenues of Research if ported to a larger scale

As an example,

MACHINE LEARNING (ML)

- has provided elegant and sophisticated solutions to many complex problems on a small scale,
- but efforts are routinely limited by complexity and running time of SEQUENTIAL algorithms.

As an example,

MACHINE LEARNING (ML)

- has provided elegant and sophisticated solutions to many complex problems on a small scale,
- but efforts are routinely limited by complexity and running time of SEQUENTIAL algorithms.

described as,

a community full of "ENTRENCHED PROCEDURAL PROGRAMMERS"

typically focus on optimizing sequential algorithms when faced with scaling problems.

As an example,

MACHINE LEARNING (ML)

- has provided elegant and sophisticated solutions to many complex problems on a small scale,
- but efforts are routinely limited by complexity and running time of SEQUENTIAL algorithms.

described as,

a community full of "ENTRENCHED PROCEDURAL PROGRAMMERS"

typically foc hen faced
with scalin

need to make it easier to experiment with parallelism

What about MapReduce?

What about MapReduce?

Poor support for iteration.

MapReduce instances must be chained together in order to achieve iteration.

- Not always straightforward.
  Even building non-cyclic pipelines is hard (e.g., FlumeJava, PLDI'10).
- Overhead is significant.
  Communication, serialization (e.g., Phoenix, IISWC'09).

Menthor...

Menthor...

- is a framework for parallel graph processing.
  (But it is not limited to graphs.)

Menthor...

- is a framework for parallel graph processing.
  (But it is not limited to graphs.)
- is inspired by BSP.
  With functional reduction/aggregation mechanisms.

Menthor...

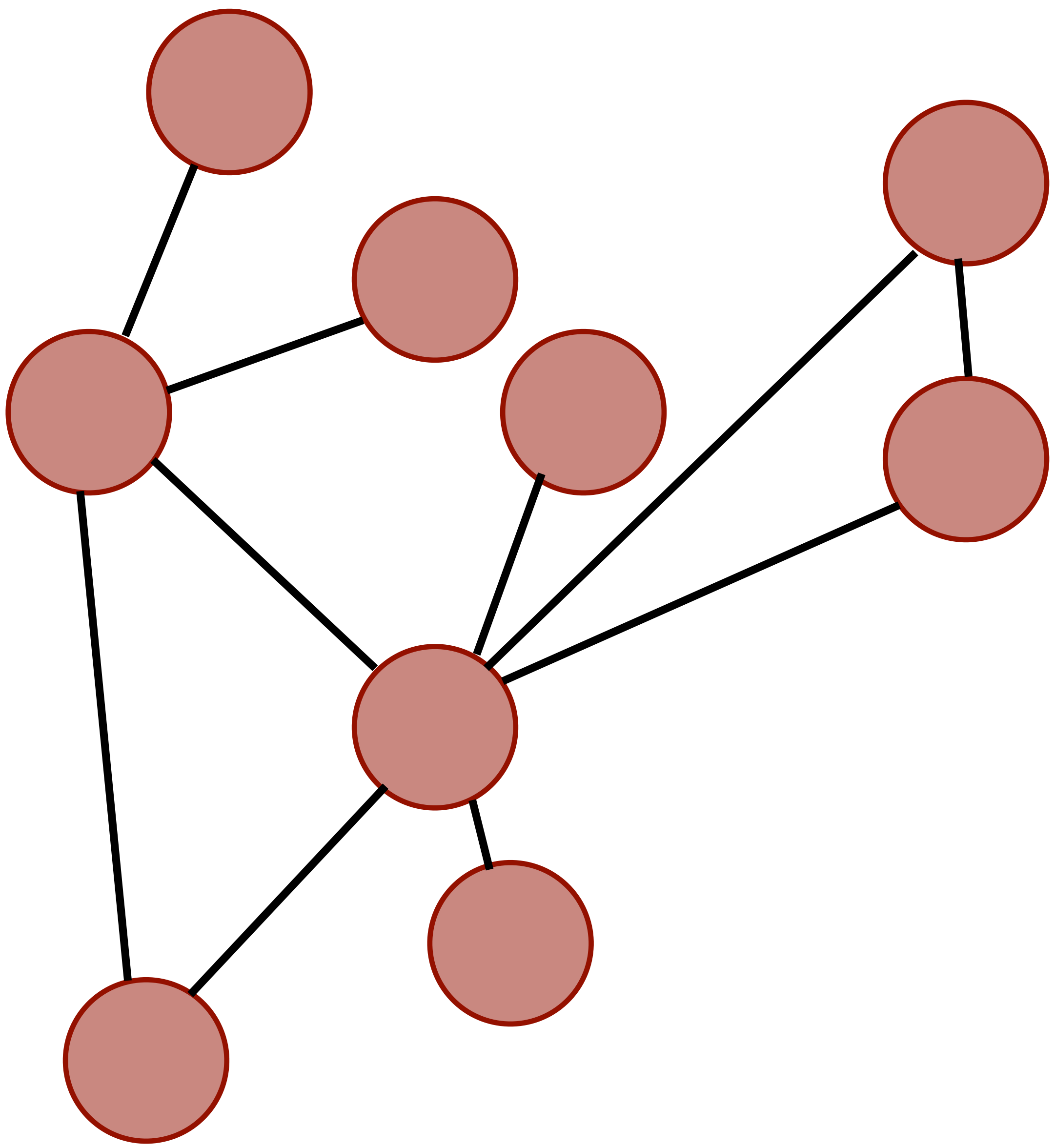
- is a framework for parallel graph processing.
  (But it is not limited to graphs.)
- is inspired by BSP.
  With functional reduction/aggregation mechanisms.
- avoids an inversion of control
  of other BSP-inspired graph-processing frameworks.

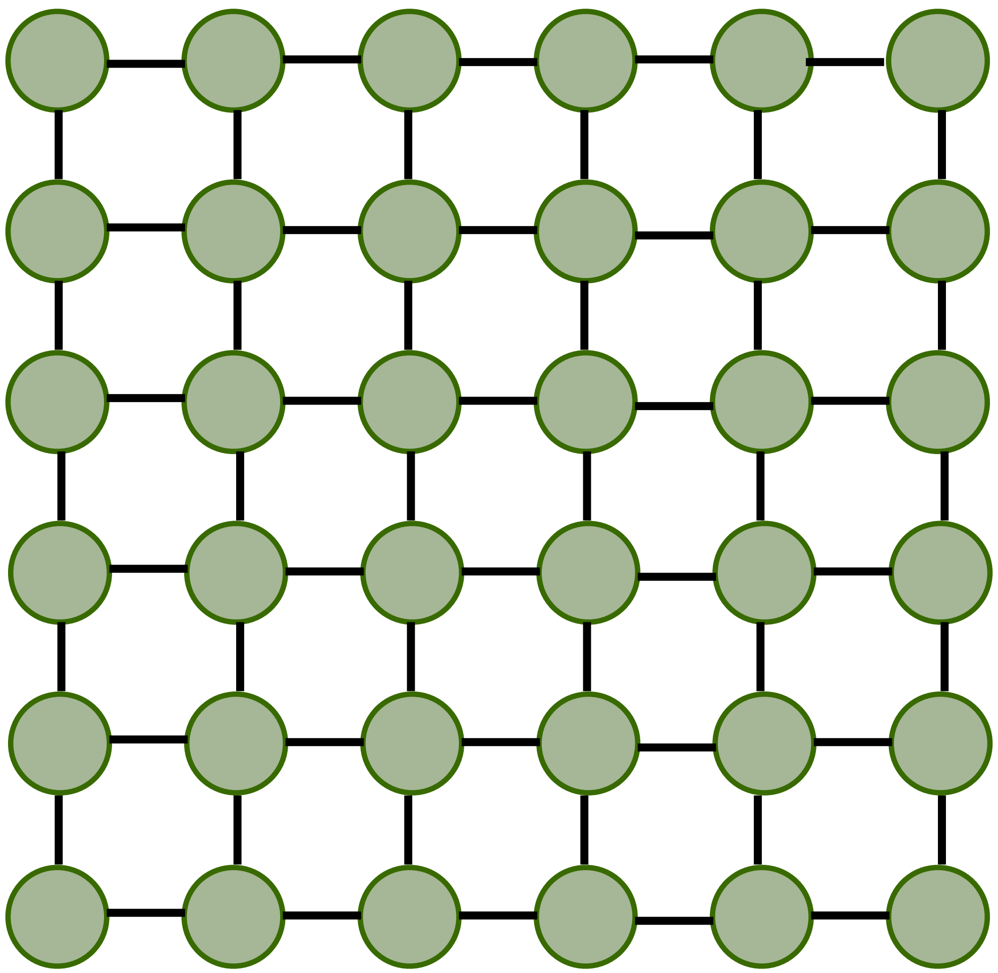
Menthor...

- is a framework for parallel graph processing.
  (But it is not limited to graphs.)
- is inspired by BSP.
  With functional reduction/aggregation mechanisms.
- avoids an inversion of control
  of other BSP-inspired graph-processing frameworks.
- is implemented in Scala,
  and there is a preliminary experimental evaluation.

Menthor's
Model of Computation.

Data.

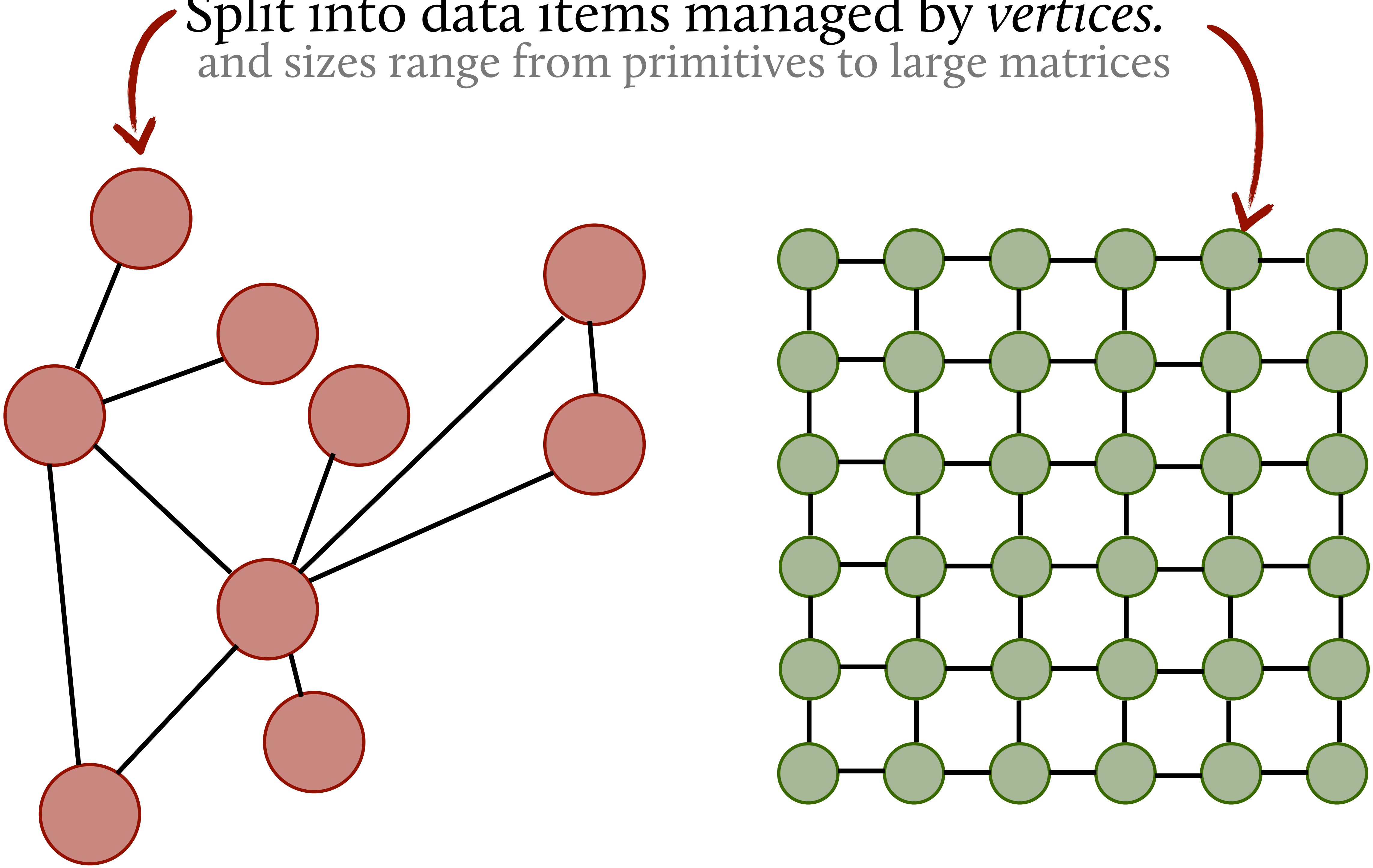




Data.

Split into data items managed by *vertices.*

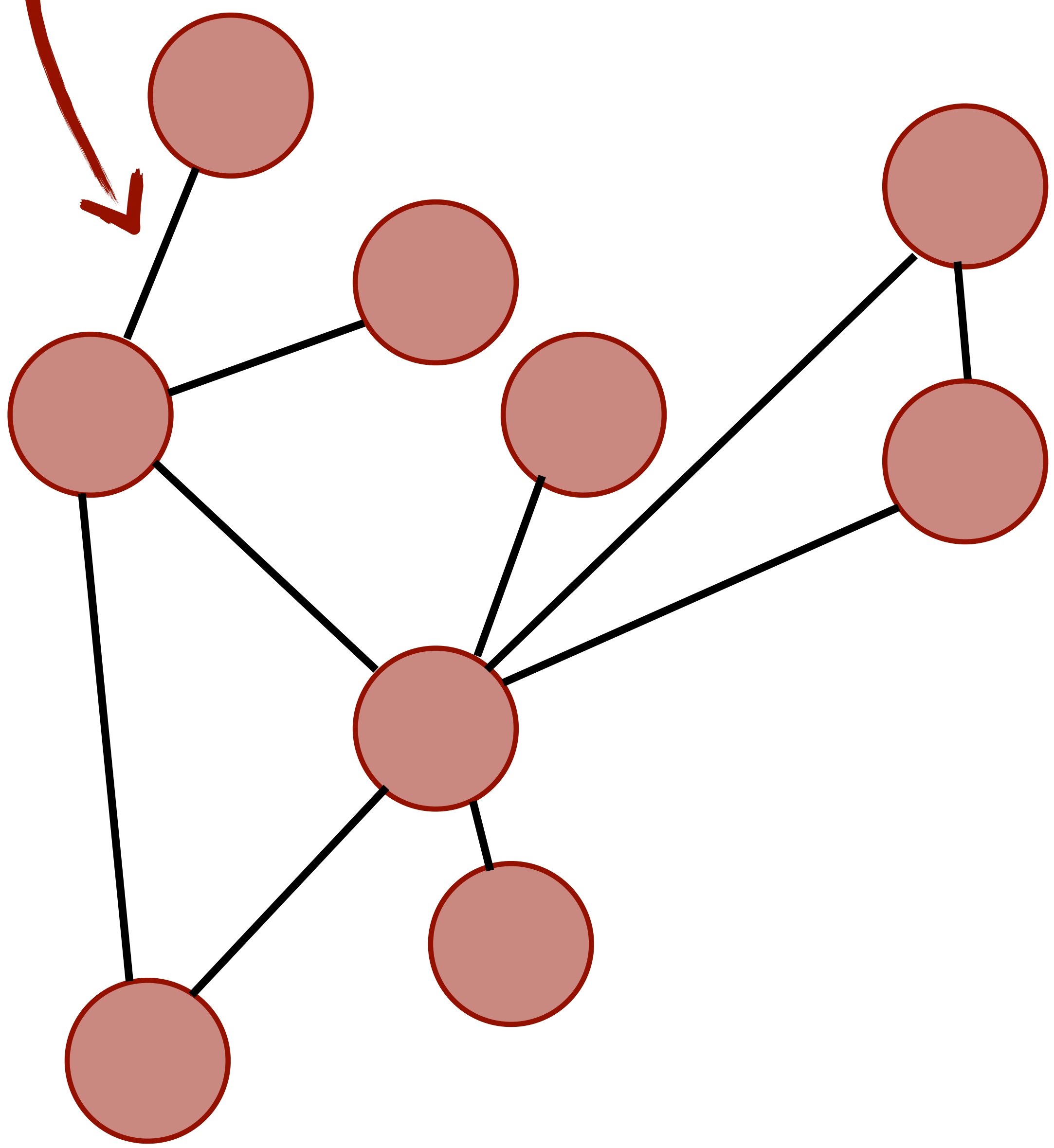
and sizes range from primitives to large matrices

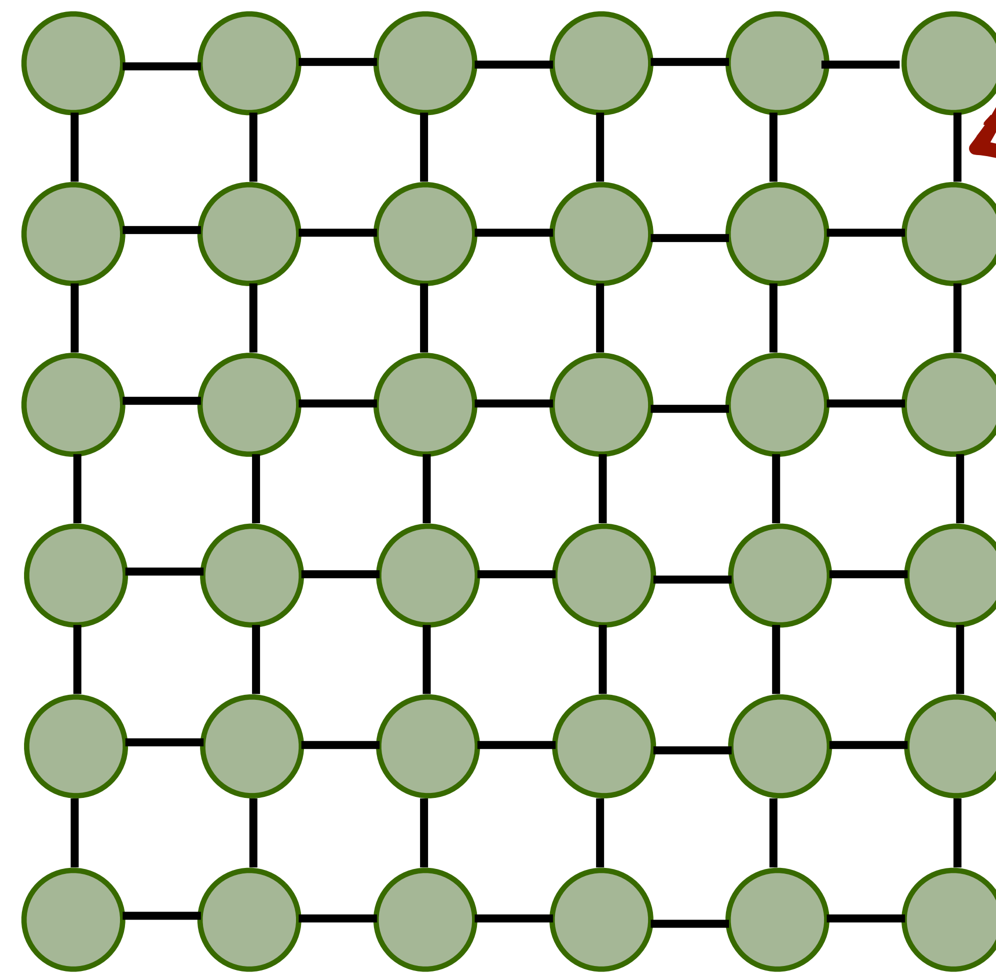


Data.

Split into data items managed by *vertices.*

Relationships expressed using *edges* between vertices.





Algorithms.

Algorithms.

- Data items stored inside of vertices *iterativelу* updated.

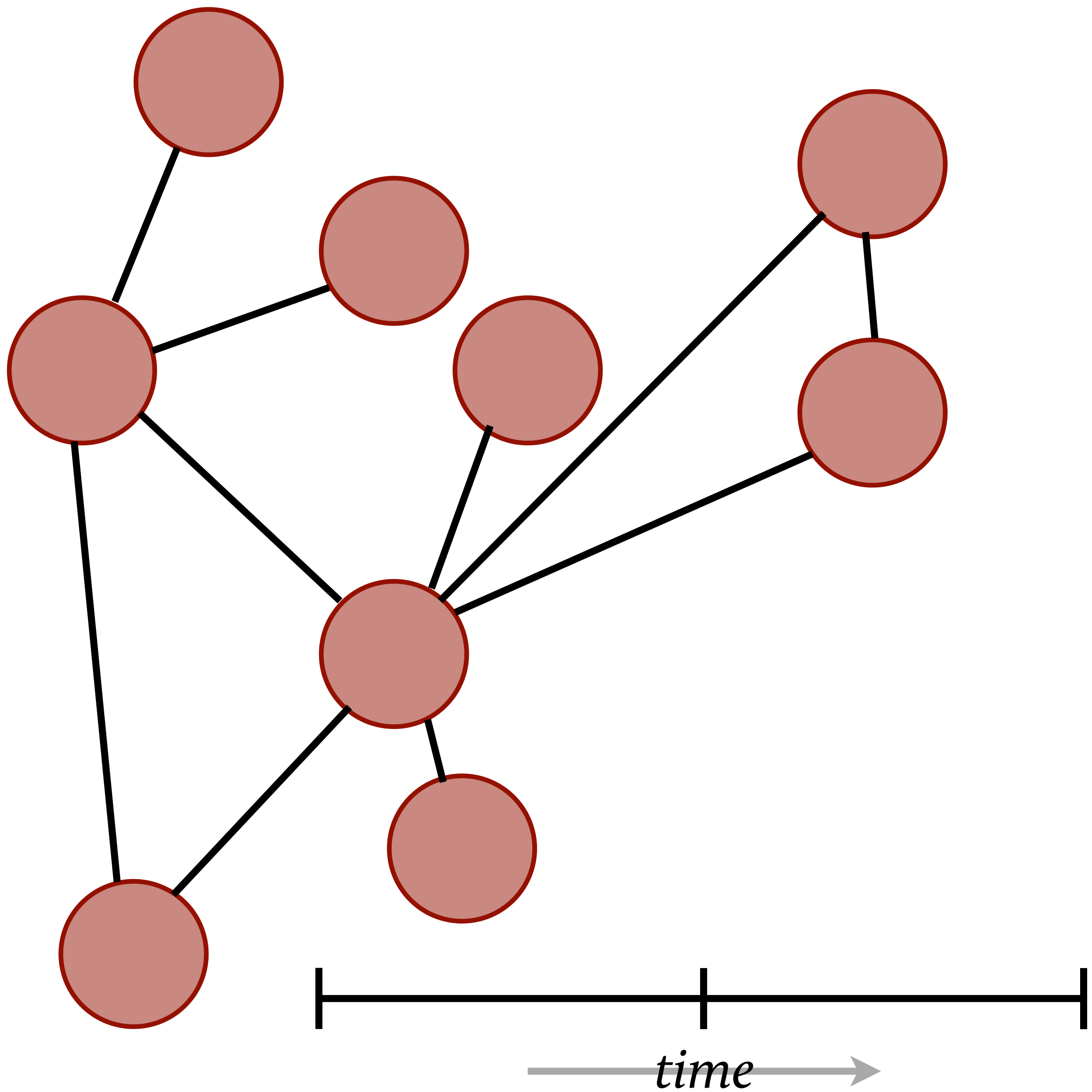
Algorithms.

- Data items stored inside of vertices *iteratively* updated.
- Iterations happen as **SYNCHRONIZED SUPERSTEPS.**

(inspired by the BSP model)

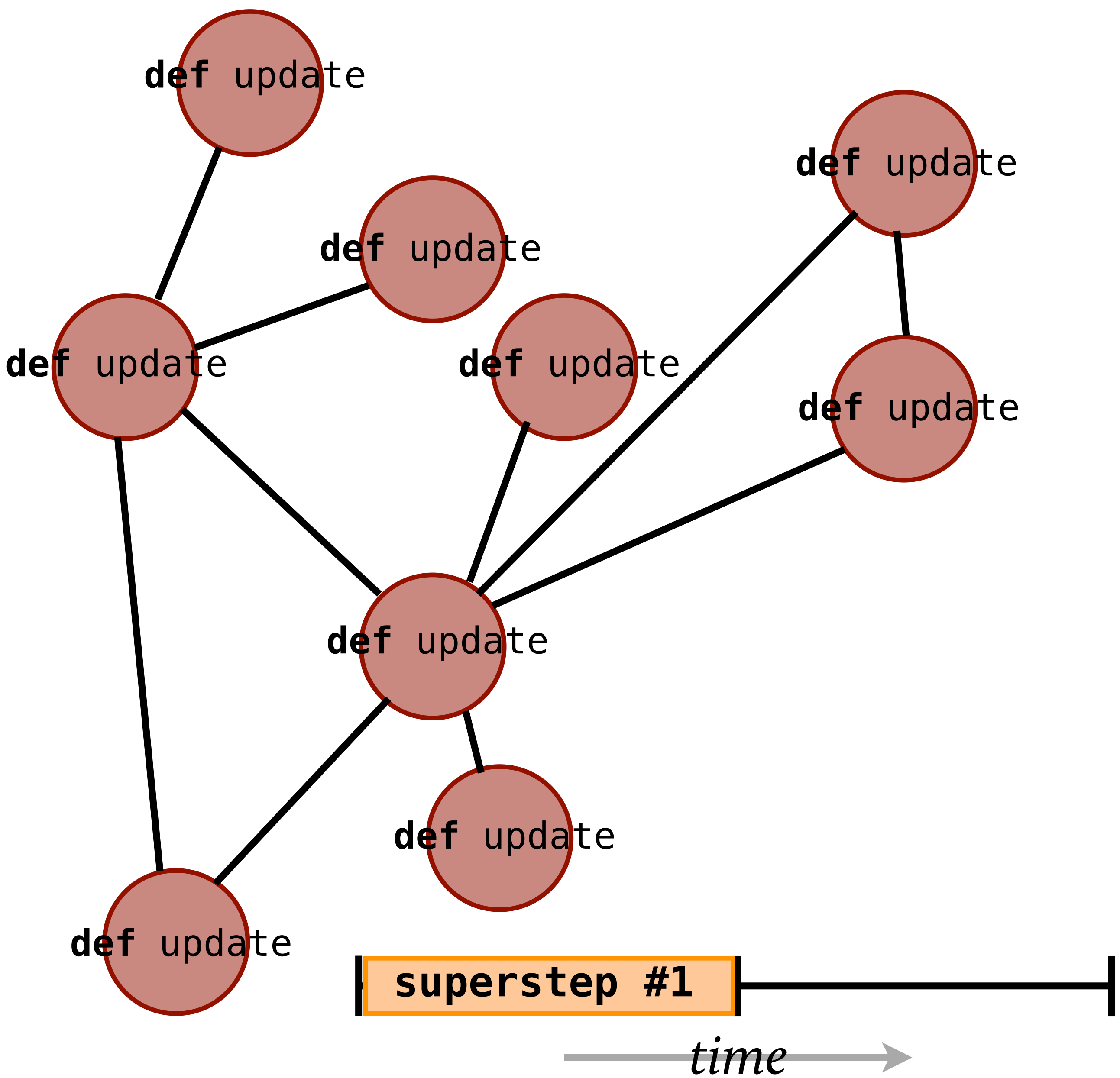
Algorithms.

- Data items stored inside of vertices *iteratively* updated.
- Iterations happen as **SYNCHRONIZED SUPERSTEPS.**



Algorithms.

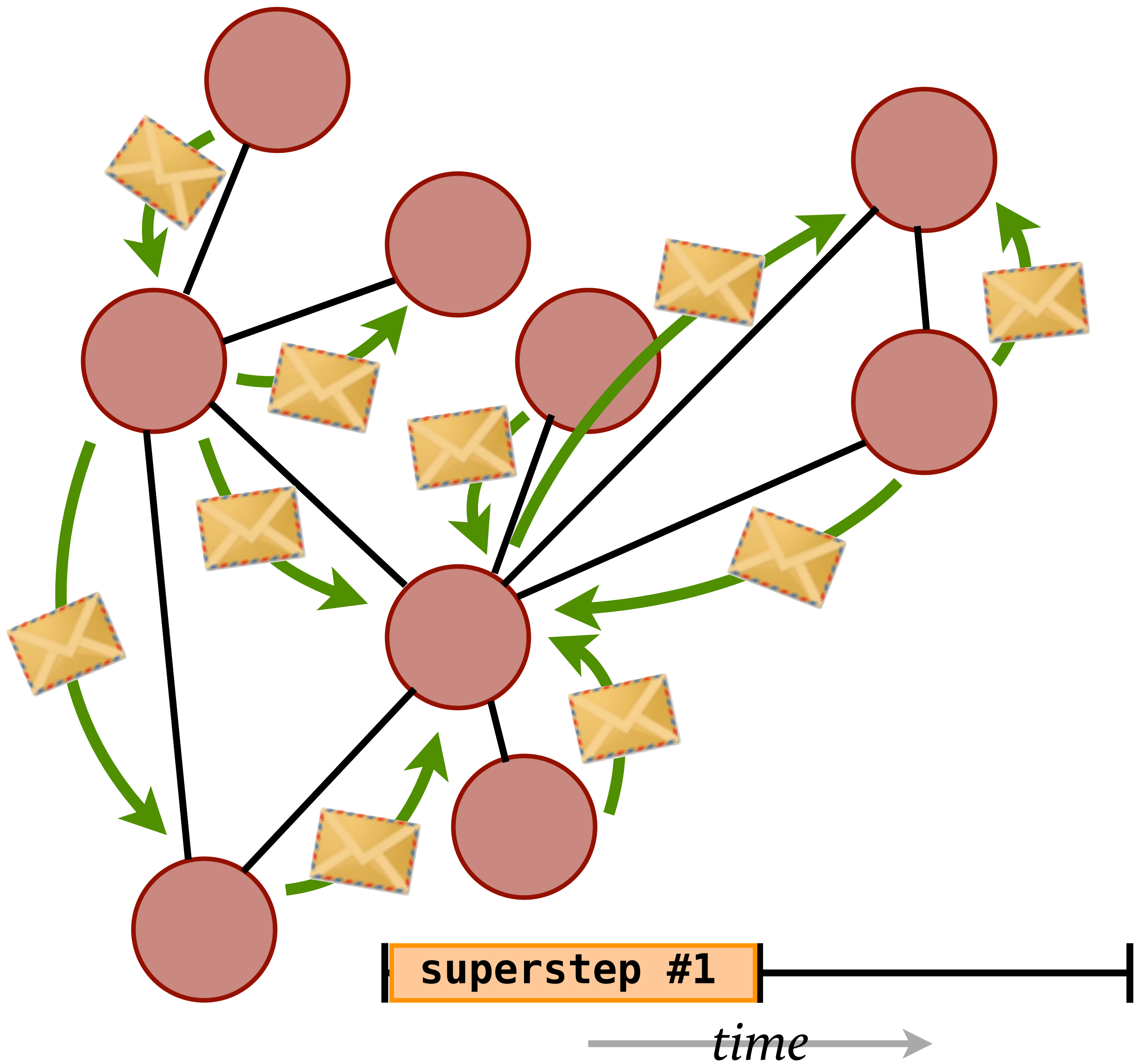
- Data items stored inside of vertices *iteratively* updated.
- Iterations happen as **SYNCHRONIZED SUPERSTEPS.**



I. | update each vertex in *parallel.*

Algorithms.

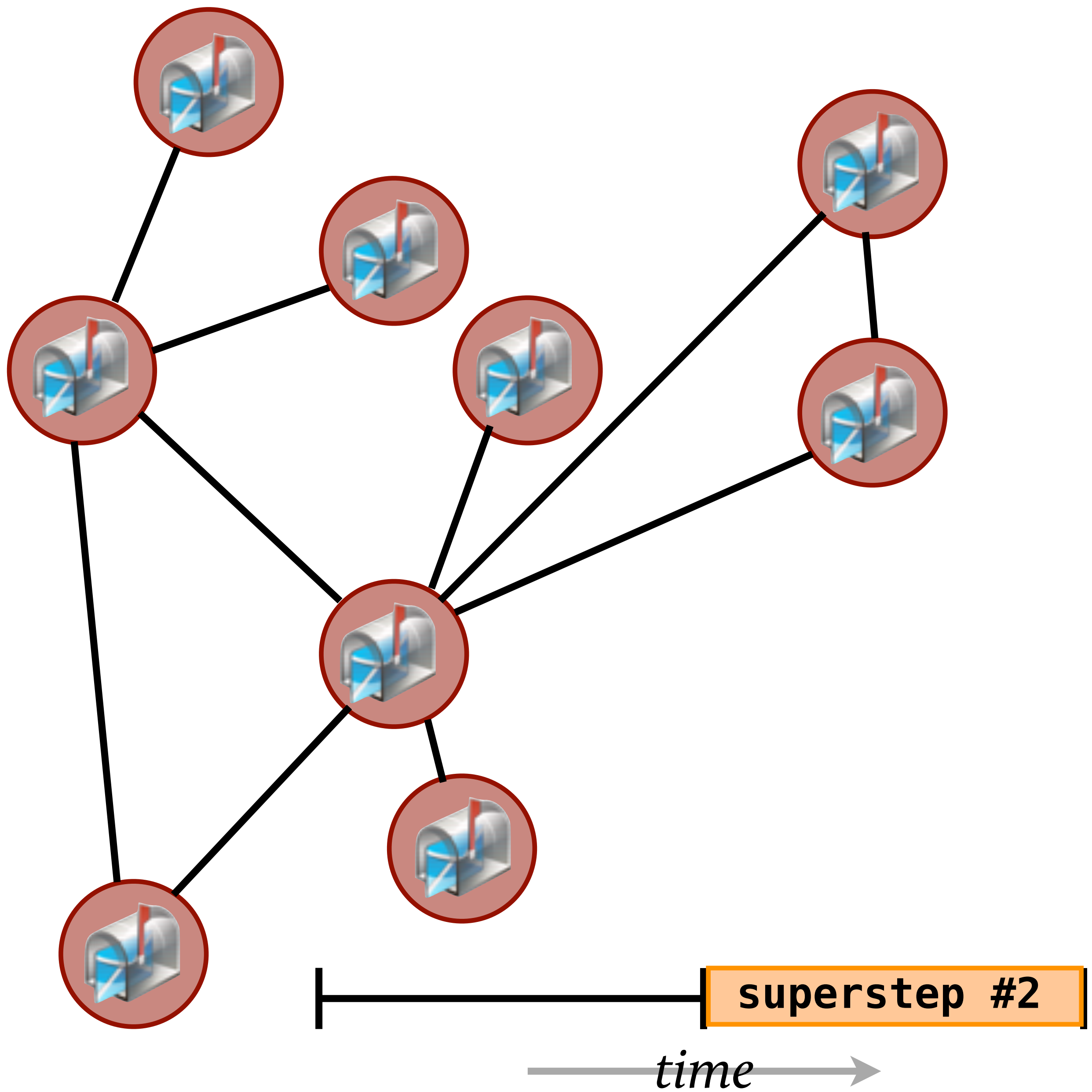
- Data items stored inside of vertices *iteratively* updated.
- Iterations happen as **SYNCHRONIZED SUPERSTEPS.**



1. update each vertex in *parallel.*
2. `update` produces *outgoing* messages to other vertices

Algorithms.

- Data items stored inside of vertices *iteratively* updated.
- Iterations happen as **SYNCHRONIZED SUPERSTEPS.**



1. update each vertex in *parallel.*
2. `update` produces *outgoing* messages to other vertices
3. incoming messages available at the beginning of the next **SUPERSTEP.**

Substeps. (and Messages)

SUBSTEPS are computations that,

Substeps. (and Messages)

SUBSTEPS are computations that,

1. update the value of `this Vertex`

Substeps. (and Messages)

SUBSTEPS are computations that,

1. update the value of `this Vertex`
2. return a list of messages:

```
case class Message[Data](source: Vertex[Data],
  dest: Vertex[Data], value: Data)
```

Substeps. (and Messages)

SUBSTEPS are computations that,

1. update the value of `this Vertex`
2. return a list of messages:

```
case class Message[Data](source: Vertex[Data],
  dest: Vertex[Data], value: Data)
```

EXAMPLES...

```
{
  value = ...
  List()
}
```

Substeps. (and Messages)

Substeps are computations that,

1. update the value of `this Vertex`
2. return a list of messages:

```
case class Message[Data](source: Vertex[Data],
  dest: Vertex[Data], value: Data)
```

Examples...

```
{
  value = ...
  List()
}
```

```
{
  ...
  for (nb <- neighbors)
    yield Message(this, nb, value)
}
```

Substeps. (and Messages)

SUBSTEPS are computations that,

1. update the value of `this Vertex`
2. return a list of messages:

```
case class Message[Data](source: Vertex[Data],
  dest: Vertex[Data], value: Data)
```

EXAMPLES...

```
{
```

```
{
    yield Message(this, nb, value)
}
```

Each is *implicitly* converted to a `Substep[Data]`

Some Examples...

PageRank.

```
class PageRankVertex extends Vertex[Double](0.0d) {
  def update() = {
    var sum = incoming.foldLeft(0)(_ + _.value)
    value = (0.15 / numVertices) + 0.85 * sum

    if (superstep < 30) {
      for (nb <- neighbors) yield
        Message(this, nb, value / neighbors.size)
    } else
      List()
  }
}
```

Another Example.

```
class PhasedVertex extends Vertex[MyData] {
  var phase = 1

  def update() = {
    if (phase == 1) {
      ...
      if (condition)
        phase = 2
    } else if (phase == 2) {
      ...
    }
  }
}
```

Another Example.

```
class PhasedVertex extends Vertex[MyData] {
  var phase = 1
      if (condition)
        phase = 2
    } else if (phase == 2) {
      ...
    }
  }
}
```

INVERSION OF CONTROL!!
Thus, manual stack management...

Inverting the Inversion.

- Use high-level combinators to build expressions of type Substep[Data]

```
class PhasedVertex extends Vertex[MyData] {

  def update() = {
    thenUntil(condition) {
      ...
    } then {
      ...
    }
  }
}
```

Inverting the Inversion.

* Use high-level combinators to build expressions of type `Substep[Data]`

```
class PhasedVertex extends Vertex[MyData] {

  def update() = {
    thenUntil(condition) {
      ...
    } then {
      ...
    }
  }
}
```

Inverting the Inversion.

- Use high-level combinators to build expressions of type `Substep[Data]`
- Thus avoiding manual stack management.

```
class PhasedVertex extends Vertex[MyData] {

  def update() = {
    thenUntil(condition) {
      ...
    } then {
      ...
    }
  }
}
```

Reduction Combinators: `crunch` steps.

Reduction Combinators: `crunch` steps.

- Reduction operations important.
  - Replacement for shared data.
  - Global decisions.

Reduction Combinators: `crunch` steps.

- Reduction operations important.
  - Replacement for shared data.
  - Global decisions.
- Provided as just another kind of `Substep[Data]`

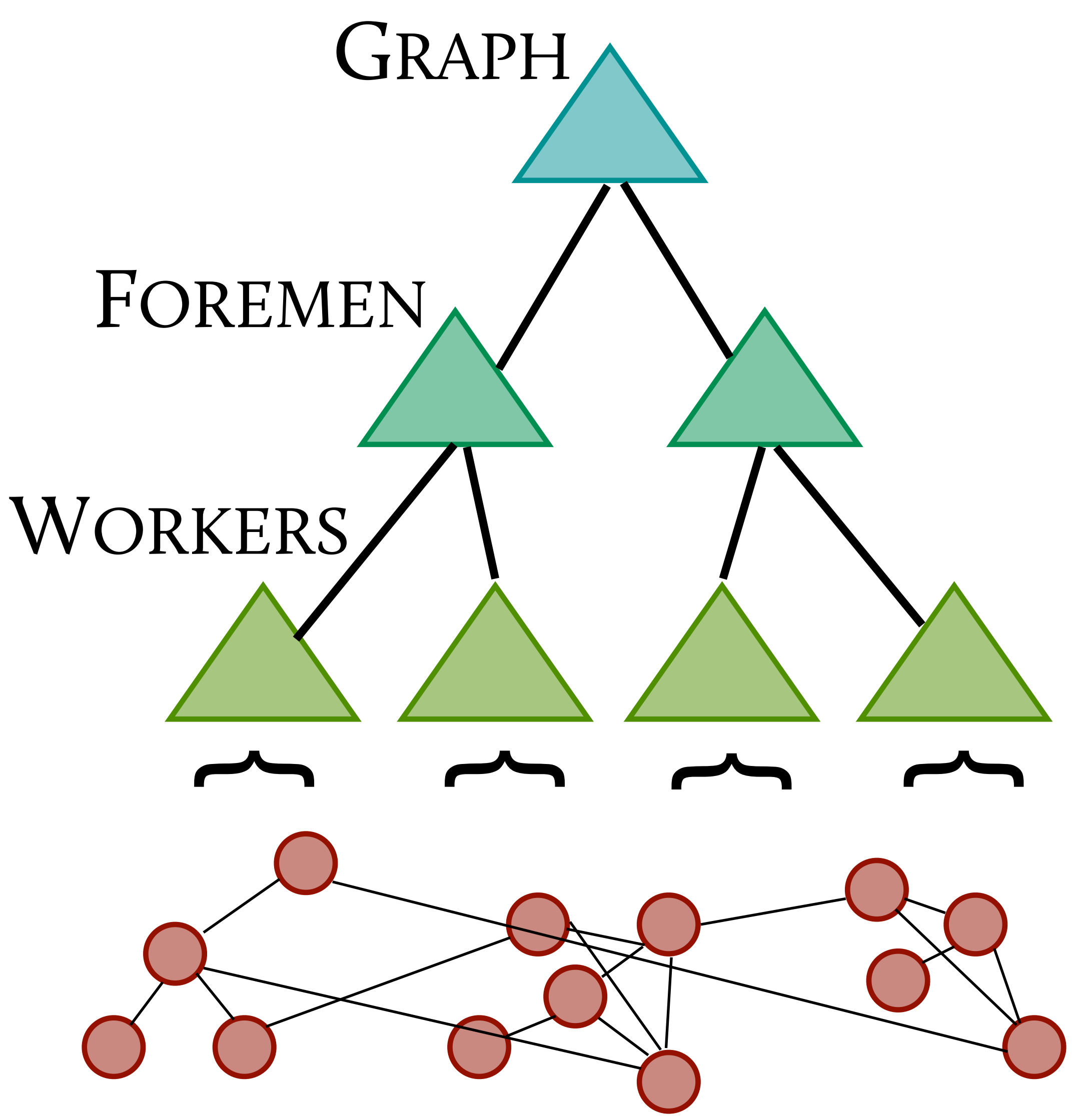
Reduction Combinators: crunch steps.

```
def update() = {
  then {
    value = ...
  } crunch ((v1: Double, v2: Double) => v1 + v2) then {
    incoming match { case List(reduced) =>
      ...
    }
  }
  ...
}
```

Menthor's
Implementation

Actors.

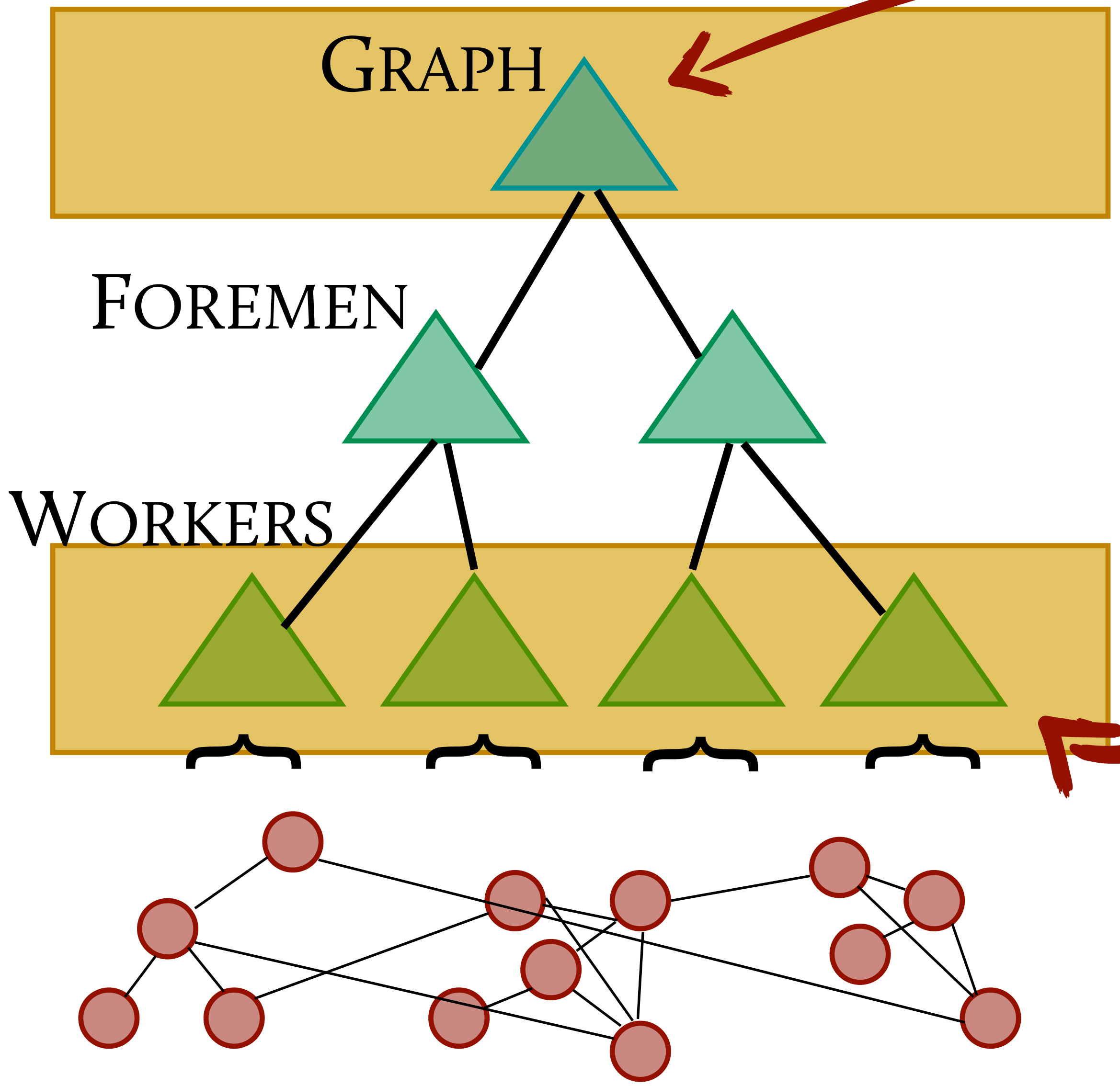
Implementation based upon Actors.



Central GRAPH instance is an actor, which manages a set of WORKER actors

Actors.

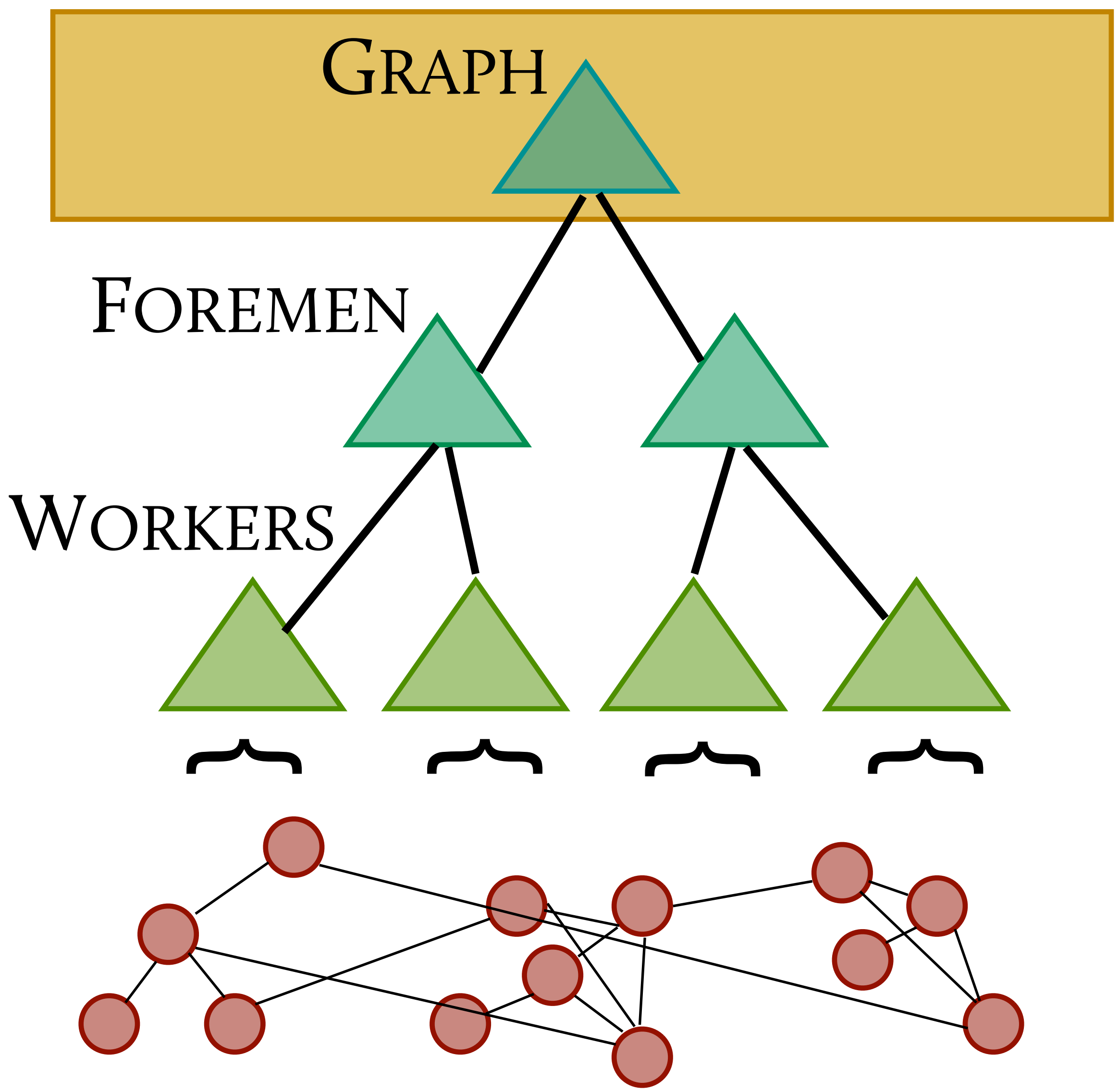
Implementation based upon Actors.



Central GRAPH instance is an actor, which manages a set of WORKER actors

Actors.

Implementation based upon Actors.

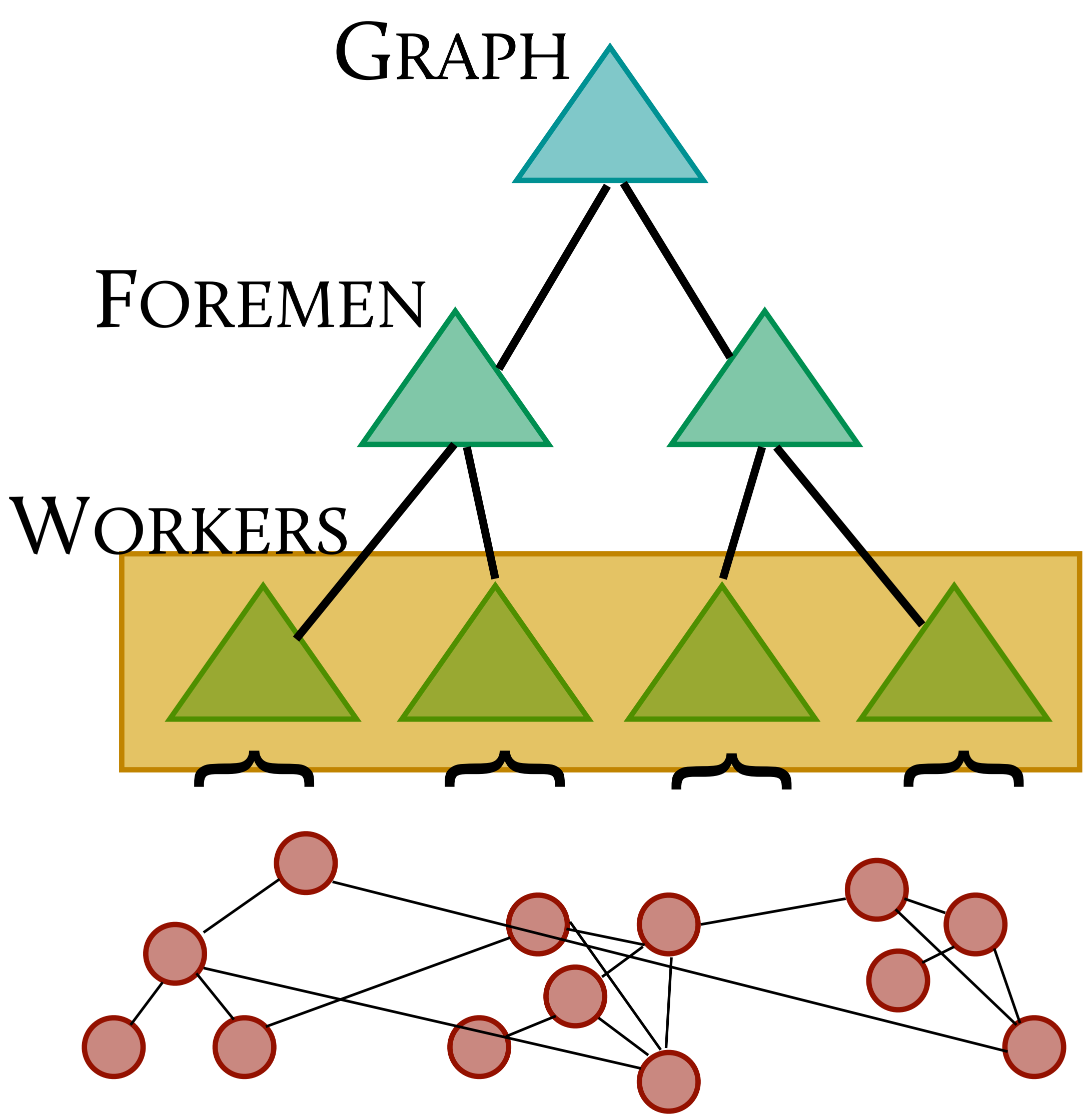


Central GRAPH instance is an actor, which manages a set of WORKER actors

GRAPH synchronizes workers using supersteps.

Actors.

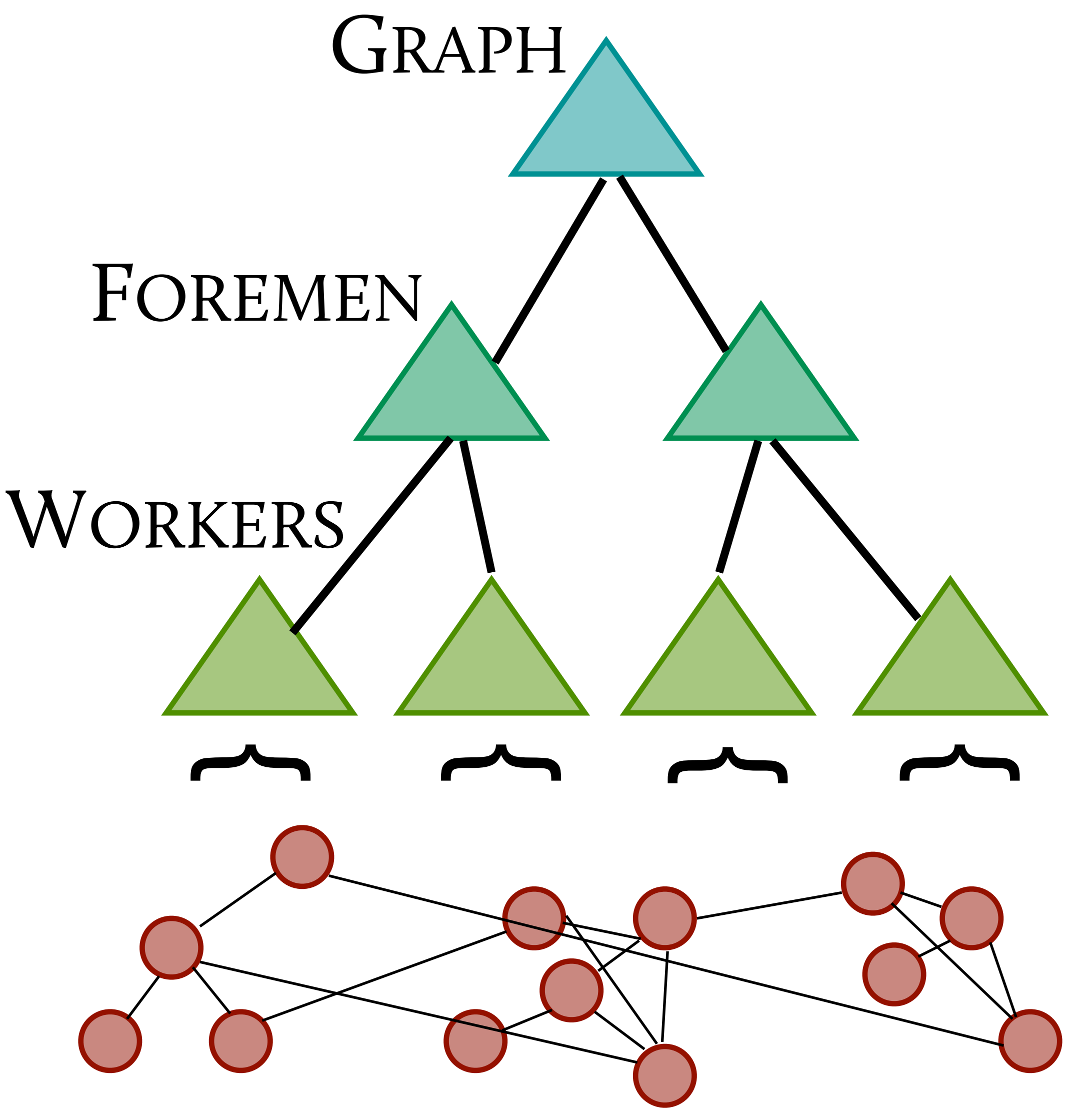
Implementation based upon Actors.



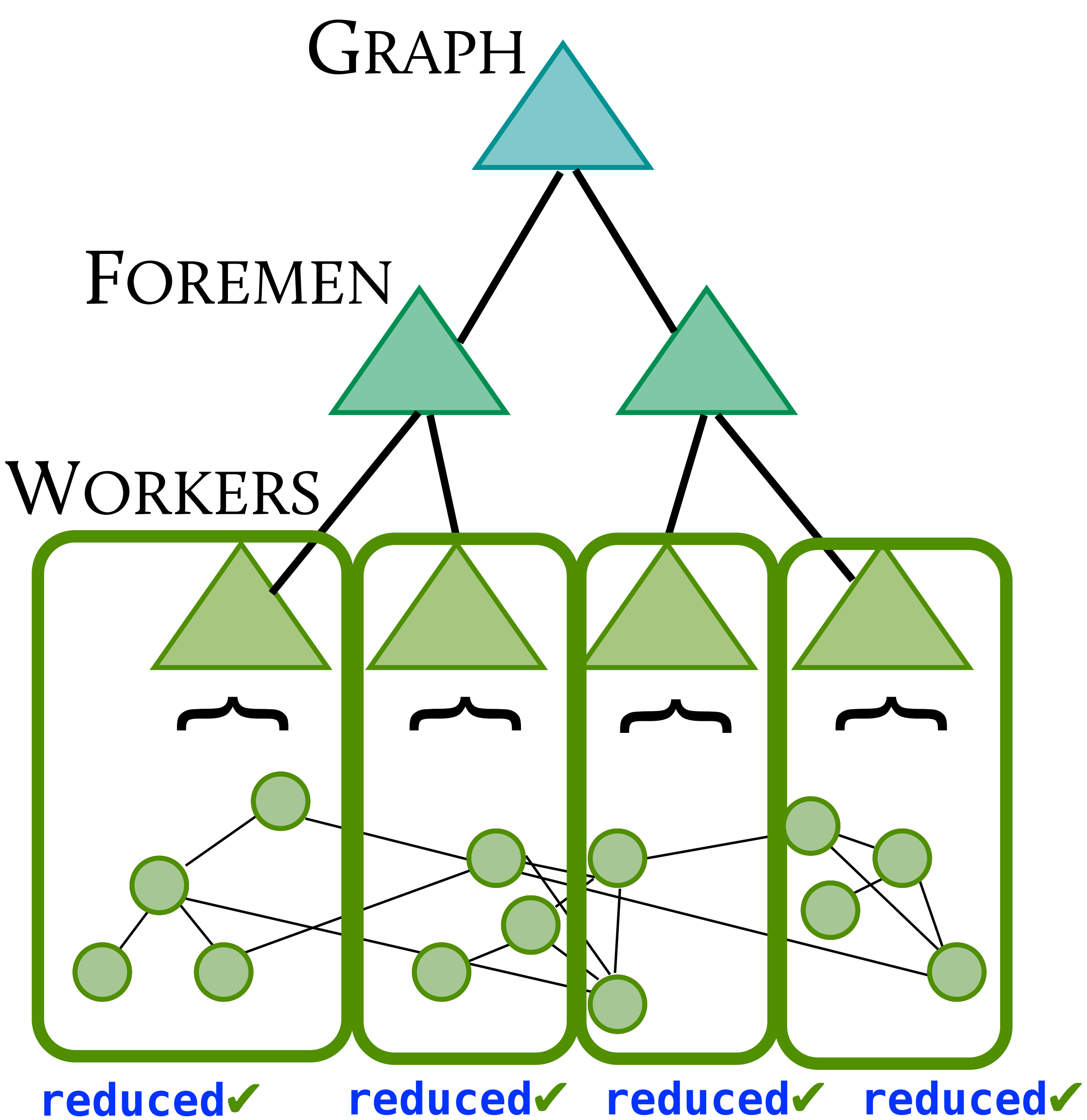
Each WORKER manages a partition of the graph's vertices,

- Deliver incoming messages that were sent in the previous superstep;
- Select and execute `update` step on each vertex in its partition;
- Forward outgoing messages generated by its vertices in the current superstep.

Implementing Reduction.

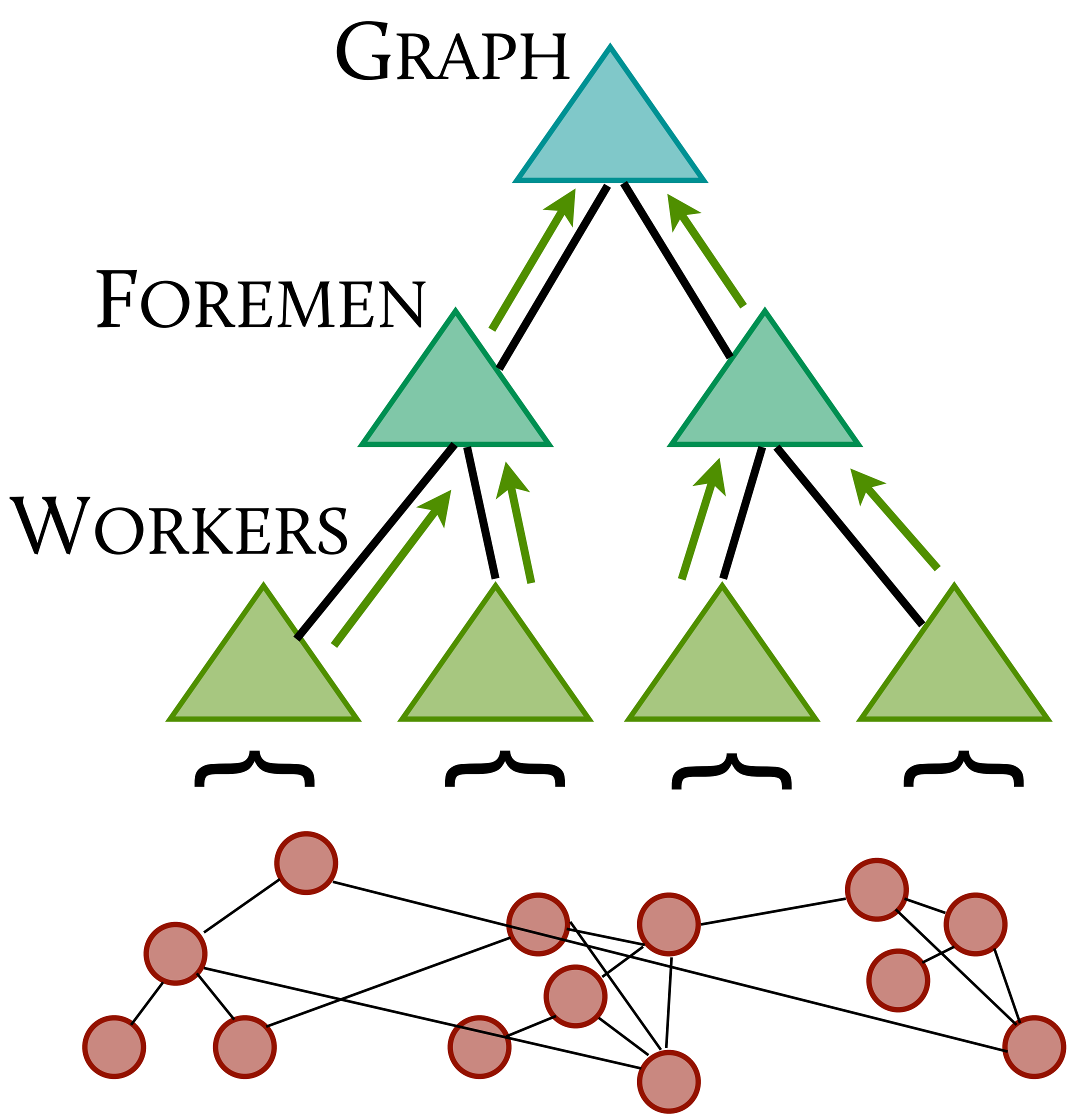


Implementing Reduction.



I. WORKER reduces the values of all vertices in its partition.

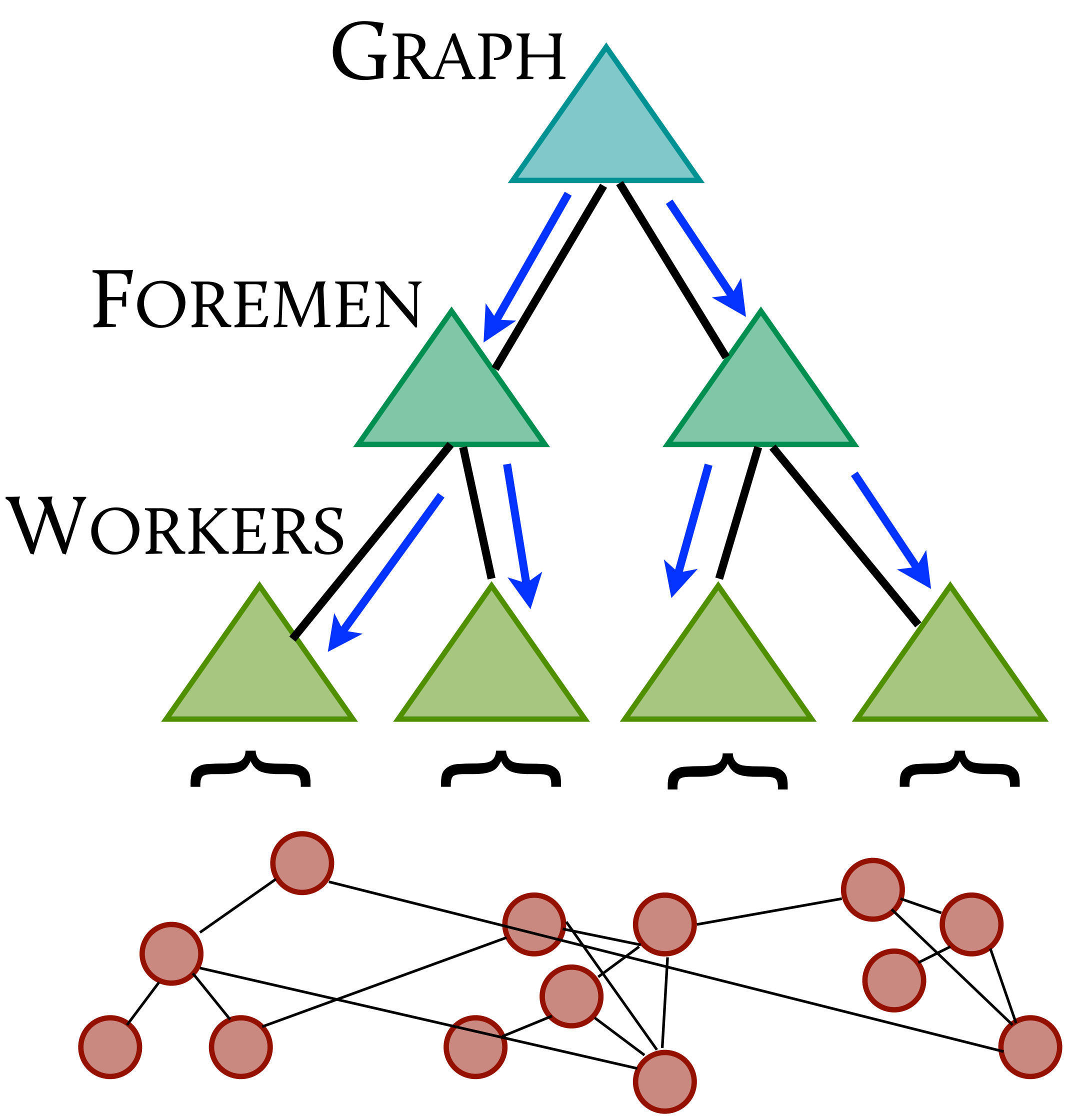
Implementing Reduction.



1. WORKER reduces the values of all vertices in its partition.

2. The result and the closure that was used to compute it is sent to the GRAPH actor, which computes the final reduced value.

Implementing Reduction.



1. WORKER reduces the values of all vertices in its partition.
2. The result and the closure that was used to compute it is sent to the GRAPH actor, which computes the final reduced value.
3. The final result is passed to all WORKERS which make it available to their vertices as incoming messages (at the beginning of the next superstep)

Implementation Principles.

Implementation Principles.

- *A pure Scala library*
  - No staging and code generation.
  - No dependency on language virtualization.

Implementation Principles.

- *A pure Scala library*
  - No staging and code generation.
  - No dependency on language virtualization.
- Benefits
  - Compatible with mainline Scala compiler.
  - Fast compilation.
  - Simple debugging and troubleshooting.
  - Framework developer-friendly.

Implementation Principles.

- *A pure Scala library*
  - No staging and code generation.
  - No dependency on language virtualization.
- Benefits
  - Compatible with mainline Scala compiler.
  - Fast compilation.
  - Simple debugging and troubleshooting.
  - Framework developer-friendly.
- Drawbacks
  - No aggressive optimizations.
  - No support for heterogeneous hardware platforms.

Related Work.

Google's Pregel
GraphLab
Signal/Collect

Main Inspiration
Graphs/BSP

Control
Inverted

Async Execution
Non-determinism

OptiML

Aggressive
Optimizations

Requires Staging

Debugging
Not optimal, yet

Spark

Designed for Iteration

Cluster support

No graph support

Non-determinism

Related Work.

Google's Pregel
GraphLab
Signal/Collect

Main Inspiration
Graphs/BSP

Control
Inverted

Async Execution
Non-determinism

OptiML

Aggressive
Optimizations

Requires Staging

Debugging
Not optimal, yet

Spark

Designed for Iteration

Cluster support

No graph support

Non-determinism

Be sure to see their talk!

Related Work.

Google's Pregel
GraphLab
Signal/Collect

Main Inspiration
Graphs/BSP

Control
Inverted

Async Execution
Non-determinism

OptiML

Aggressive
Optimizations

Requires Staging

Debugging
Not optimal, yet

Spark

Designed for Iteration

Cluster support

No graph support

Non-determinism

Be sure to see their talk!

(Many more discussed in the paper.)

Conclusions

Conclusions

- Can avoid inversion of control in vertex-based BSP using closures.

Conclusions

- Can avoid inversion of control in vertex-based BSP using closures.
- Higher-order functions useful for reductions, in an imperative model.

Conclusions

- Can avoid inversion of control in vertex-based BSP using closures.
- Higher-order functions useful for reductions, in an imperative model.
- Explicit parallelism feasible if computational model simple (cf. MapReduce)

Conclusions

- Can avoid inversion of control in vertex-based BSP using closures.
- Higher-order functions useful for reductions, in an imperative model.
- Explicit parallelism feasible if computational model simple (cf. MapReduce)
- The puzzle pieces are there to make analyzing bigger data easier.

Conclusions

- Can avoid inversion of control in vertex-based BSP using closures.
- Higher-order functions useful for reductions, in an imperative model.
- Explicit parallelism feasible if computational model simple (cf. MapReduce)
- The puzzle pieces are there to make analyzing bigger data easier.

http://lamp.epfl.ch/~phaller/menthor/

Questions?

Experimental Results.

- Applications
  - PageRank on (subset of) Wikipedia
  - Hierarchical clustering
  - Loopy belief propagation

- Very preliminary results
  - Evaluating BSP-based model
  - Implementation details changing
  - Parallel collections (extensions)