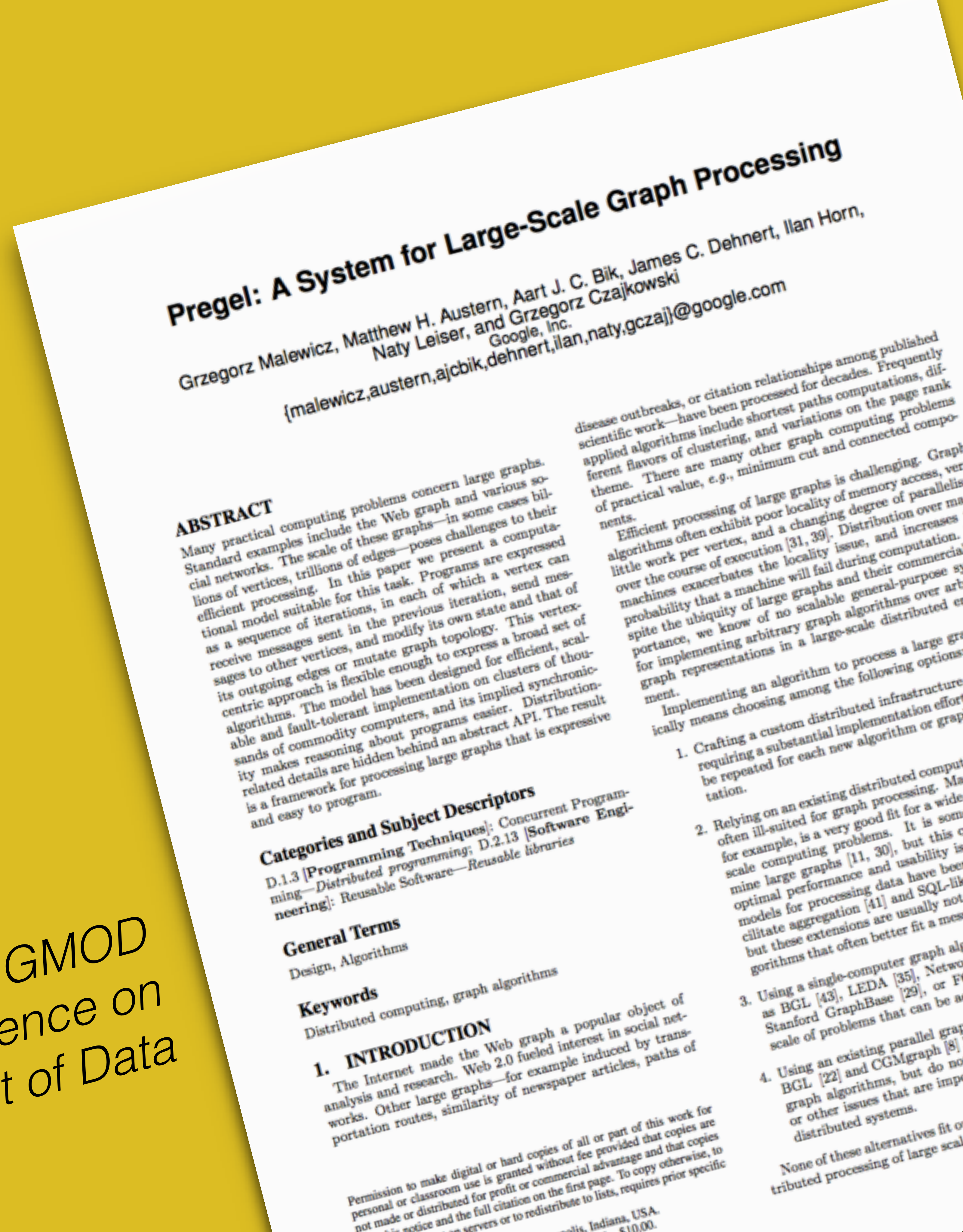
Pregel: A System for Large Scale Graph Processing

Malewicz, et.al

Presenter: Stephan Brandauer

This talk:
- High level intro



Pregel: A System for Large-Scale Graph Processing

Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski
Google, Inc.
{malewicz,austern,ajcbik,dehnert,ilan,naty,gczaj}@google.com

ABSTRACT

Many practical computing problems concern large graphs. Standard examples include the Web graph and various social networks. The scale of these graphs—in some cases billions of vertices, trillions of edges—poses challenges to their efficient processing. In this paper we present a computational model suitable for this task. Programs are expressed as a sequence of iterations, in each of which a vertex can receive messages sent in the previous iteration, send messages to other vertices, and modify its own state and that of its outgoing edges or mutate graph topology. This vertex-centric approach is flexible enough to express a broad set of algorithms. The model has been designed for efficient, scalable and fault-tolerant implementation on clusters of thousands of commodity computers, and its implied synchronicity makes reasoning about programs easier. Distribution-related details are hidden behind an abstract API. The result is a framework for processing large graphs that is expressive and easy to program.

Categories and Subject Descriptors

D.1.3 [Programming Techniques]: Concurrent Programming—*Distributed programming*; D.2.13 [Software Engineering]: Reusable Software—*Reusable libraries*

General Terms

Design, Algorithms

Keywords

Distributed computing, graph algorithms

1. INTRODUCTION

The Internet made the Web graph a popular object of analysis and research. Web 2.0 fueled interest in social networks. Other large graphs—for example induced by transportation routes, similarity of newspaper articles, paths of

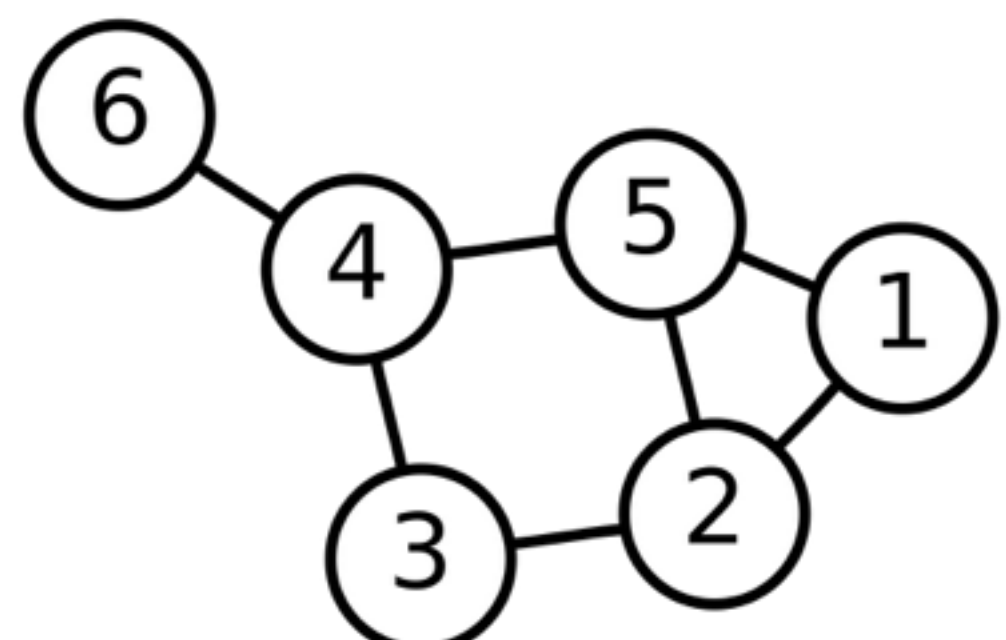
2010 ACM SIGMOD
International Conference on
Management of Data

I like high level abstractions that carefully exploit their semantics for optimisations.

Pregel is a particularly pretty case.

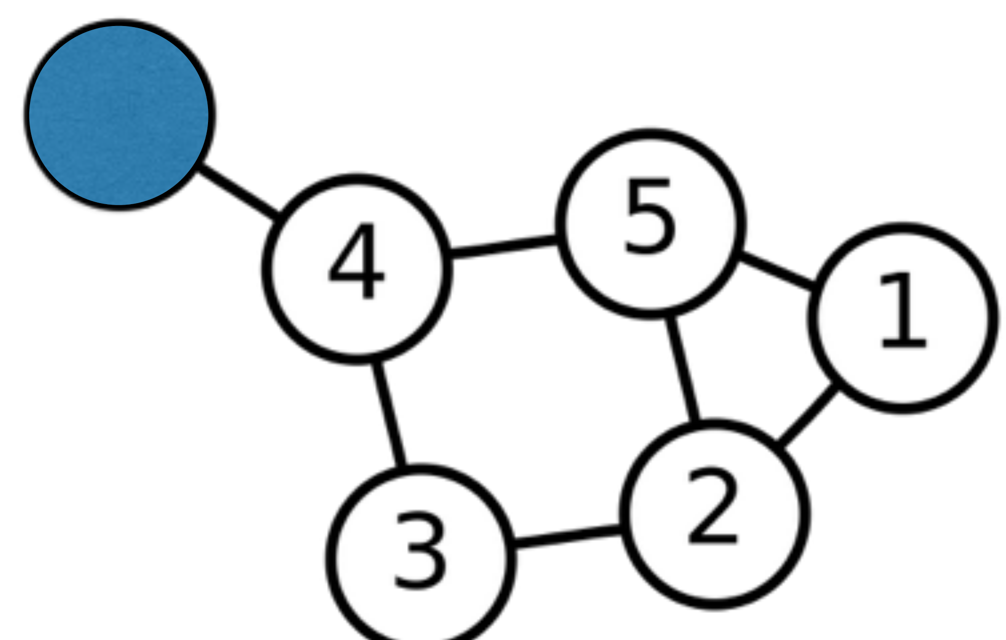
Graph Processing: hard to **parallelise**

Graphs are connected by nature — it's easy to get data races.



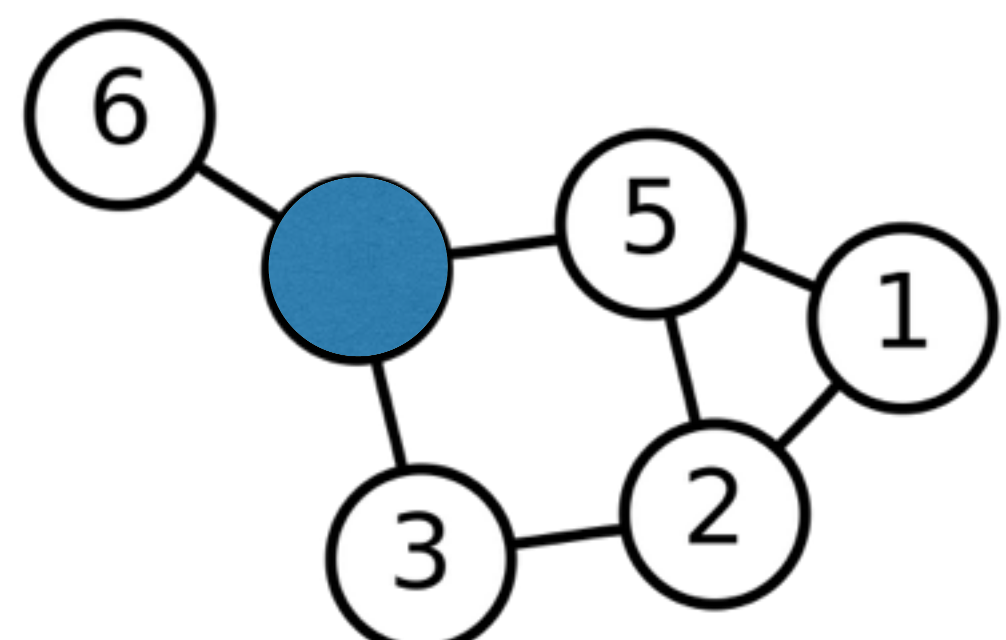
Graph Processing: hard to **parallelise**

Graphs are connected by nature — it's easy to get race conditions!



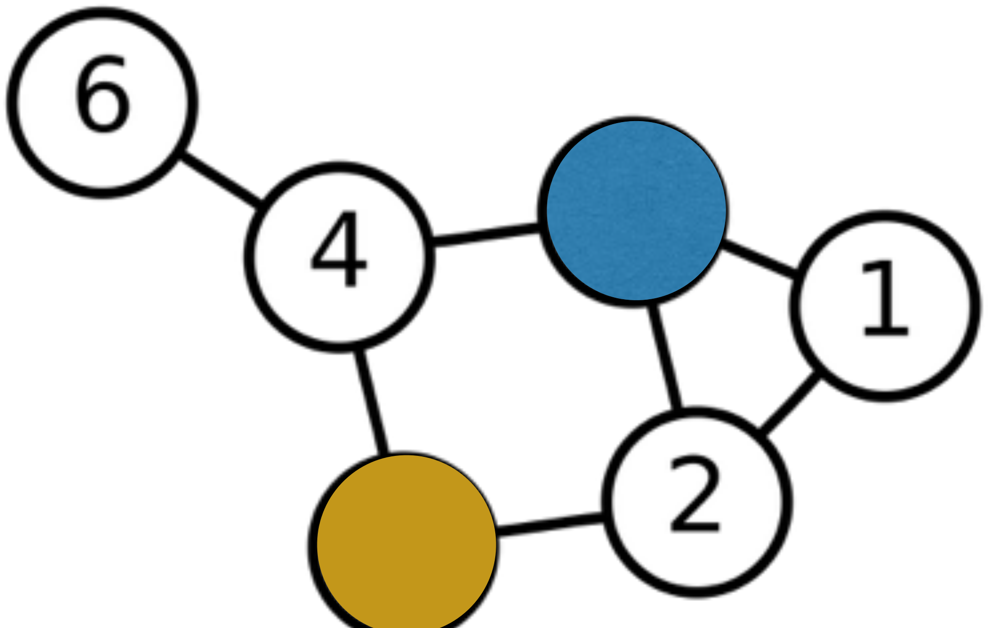
Graph Processing: hard to **parallelise**

Graphs are connected by nature — it's easy to get race conditions!



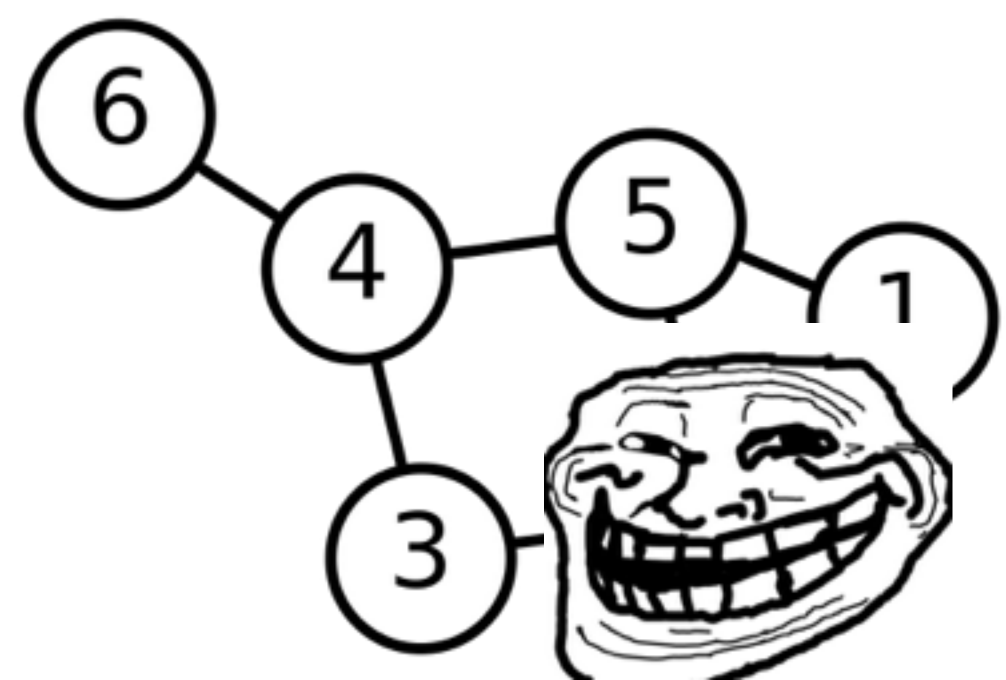
Graph Processing: hard to **parallelise**

Graphs are connected by nature — it's easy to get race conditions!



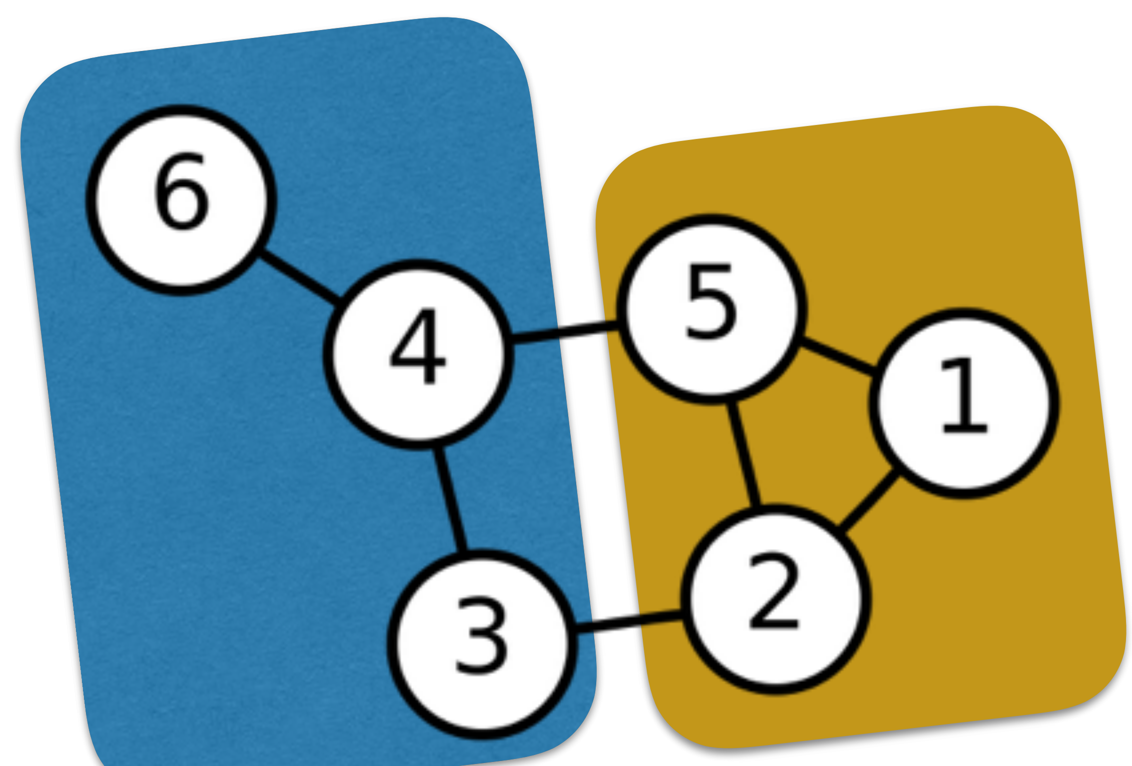
Graph Processing: hard to **parallelise**

Graphs are connected by nature — it's easy to get data races.



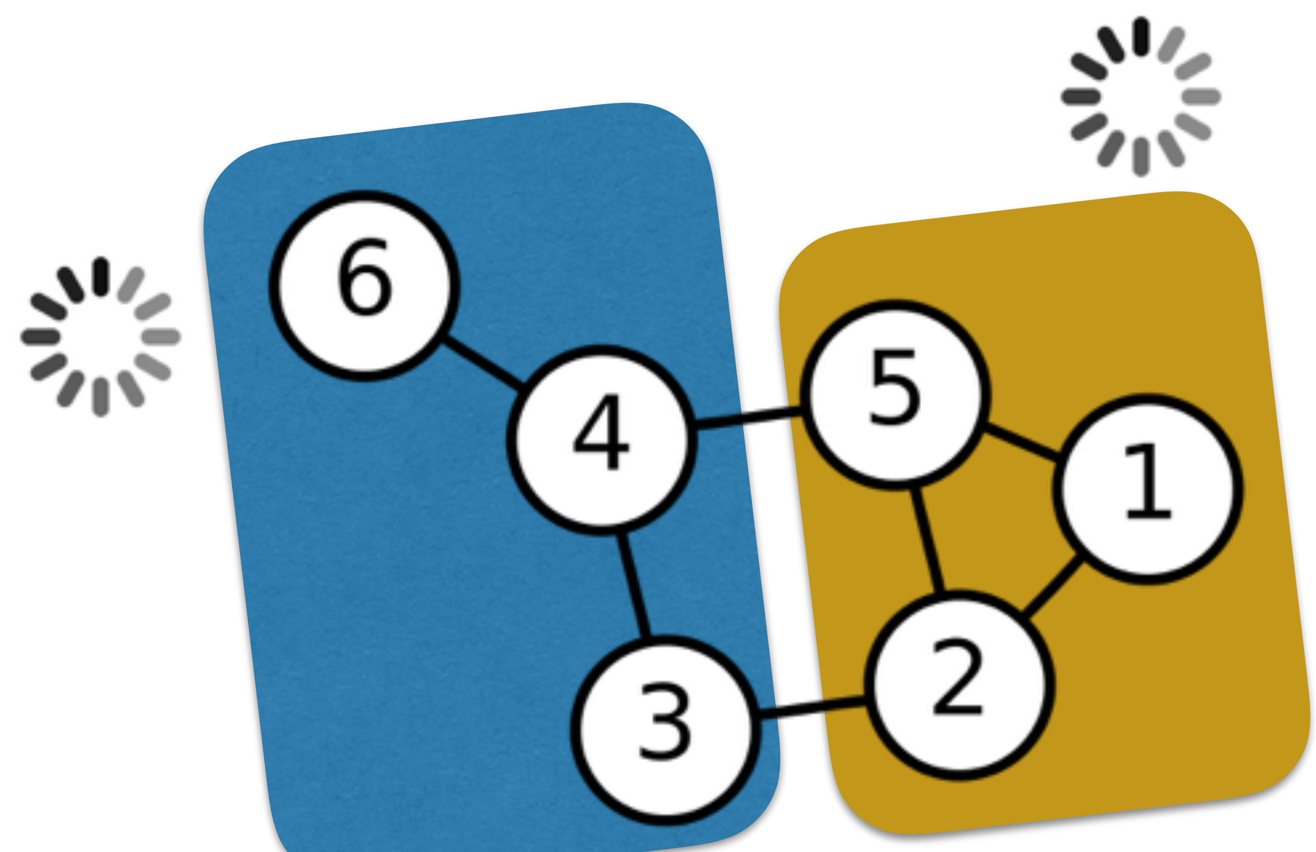
Graph Processing: hard to **distribute**

Same problems as parallelisation AND
deal with machine failures



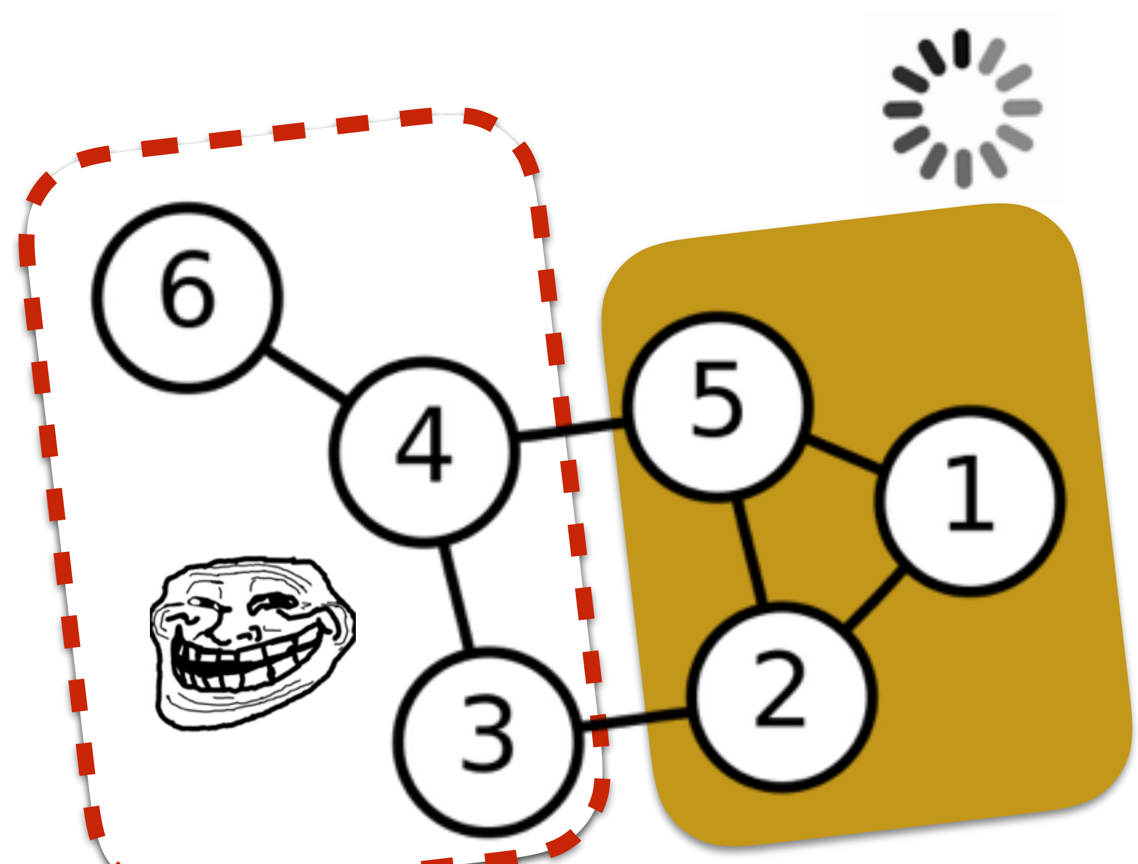
Graph Processing: hard to **distribute**

Same problems as parallelisation AND
deal with machine failures



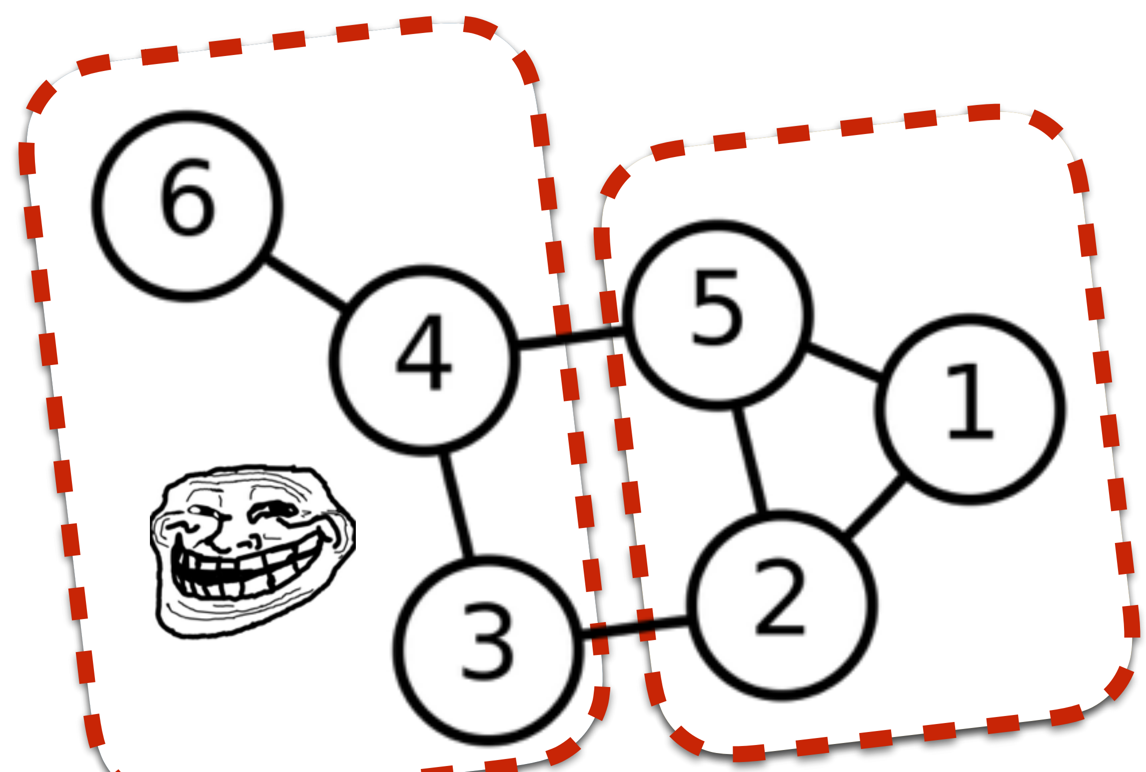
Graph Processing: hard to **distribute**

Same problems as parallelisation AND deal with machine failures



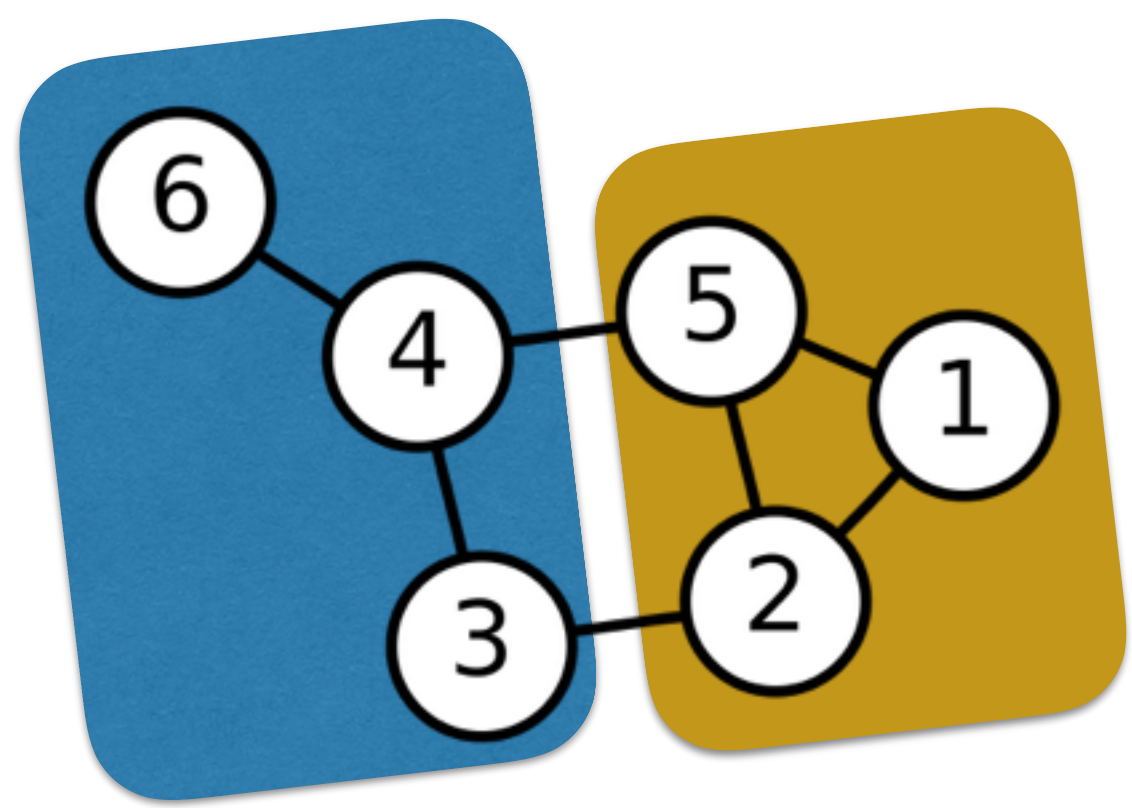
Graph Processing: hard to **distribute**

Same problems as parallelisation AND deal with machine failures



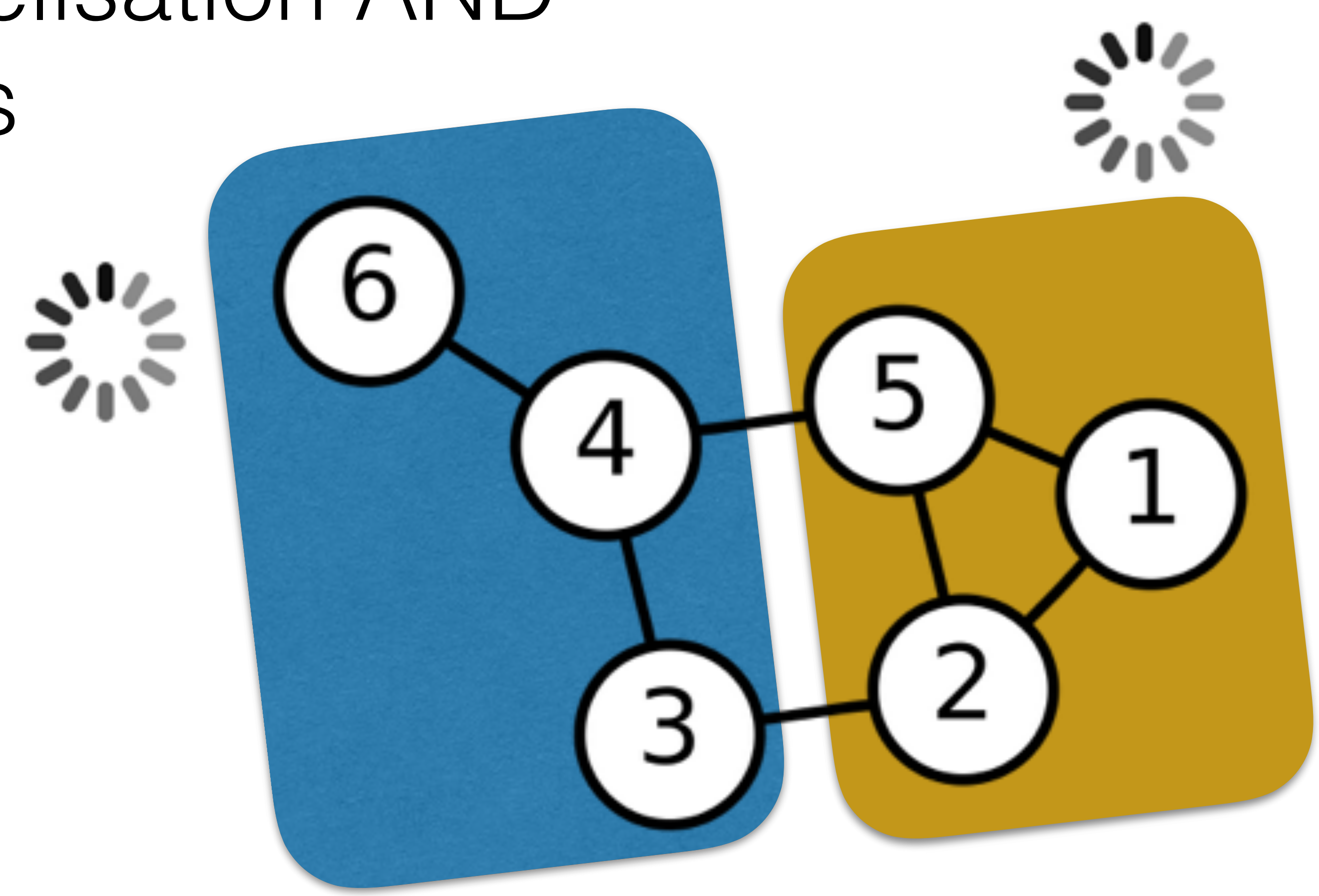
Graph Processing: hard to **distribute**

Same problems as parallelisation AND deal with machine failures



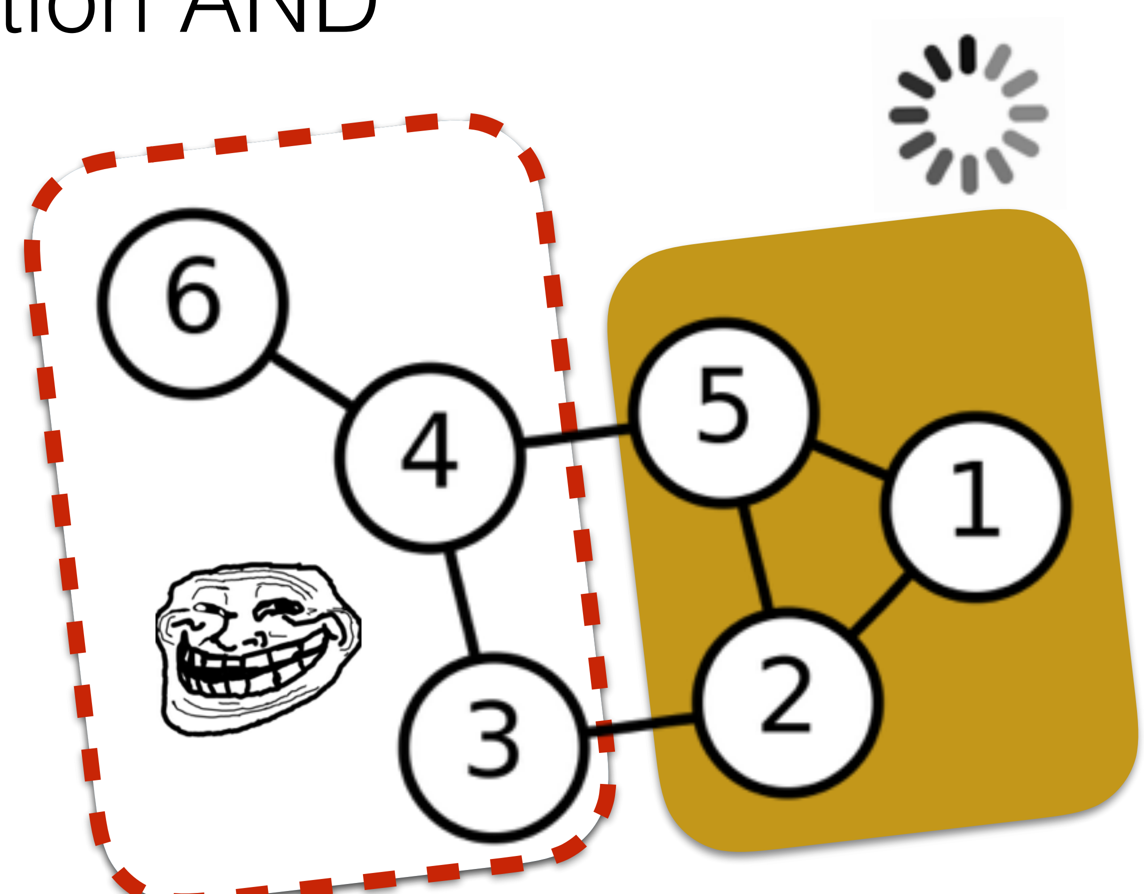
Graph Processing: hard to **distribute**

Same problems as parallelisation AND deal with machine failures



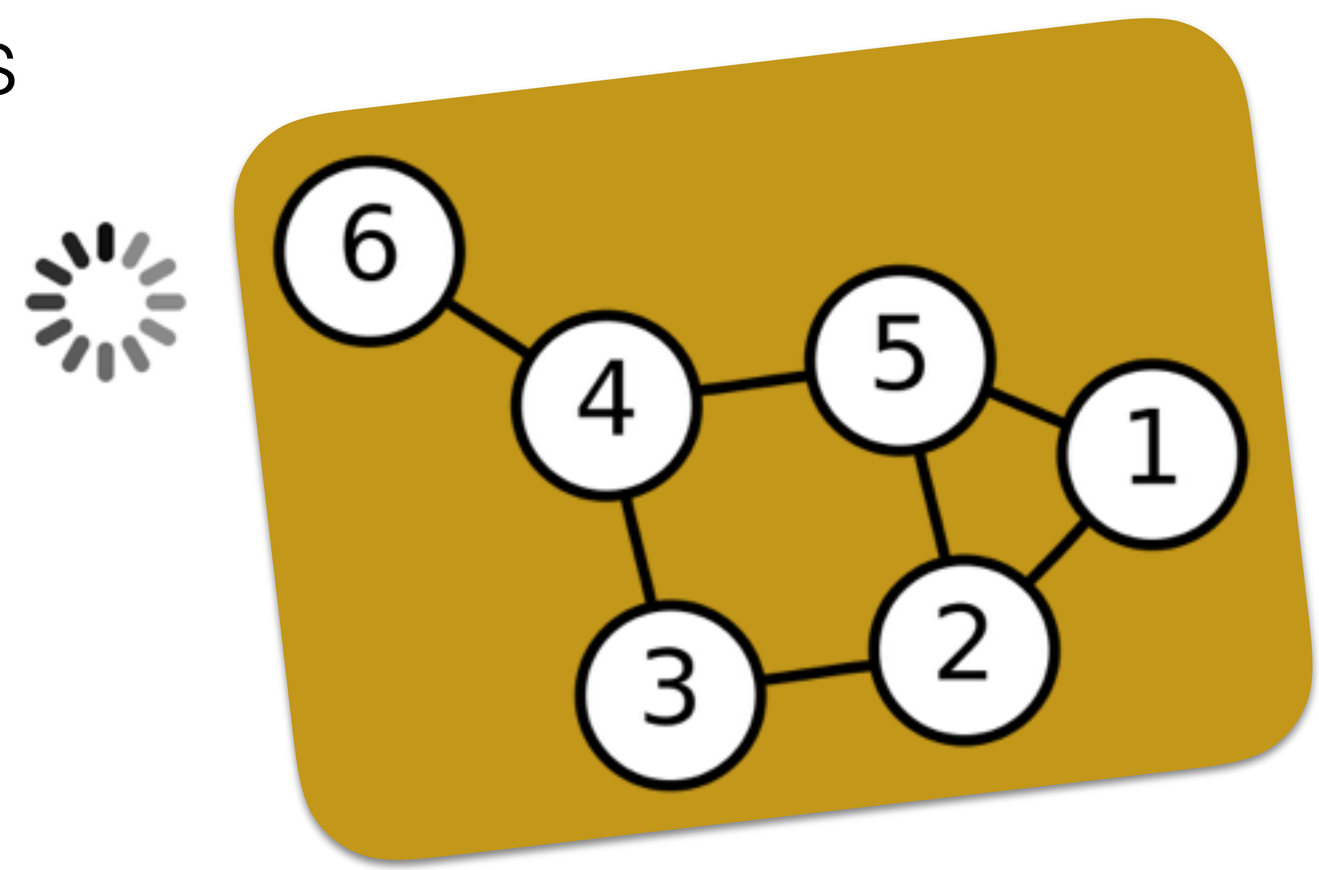
Graph Processing:
hard to **distribute**

Same problems as parallelisation AND deal with machine failures



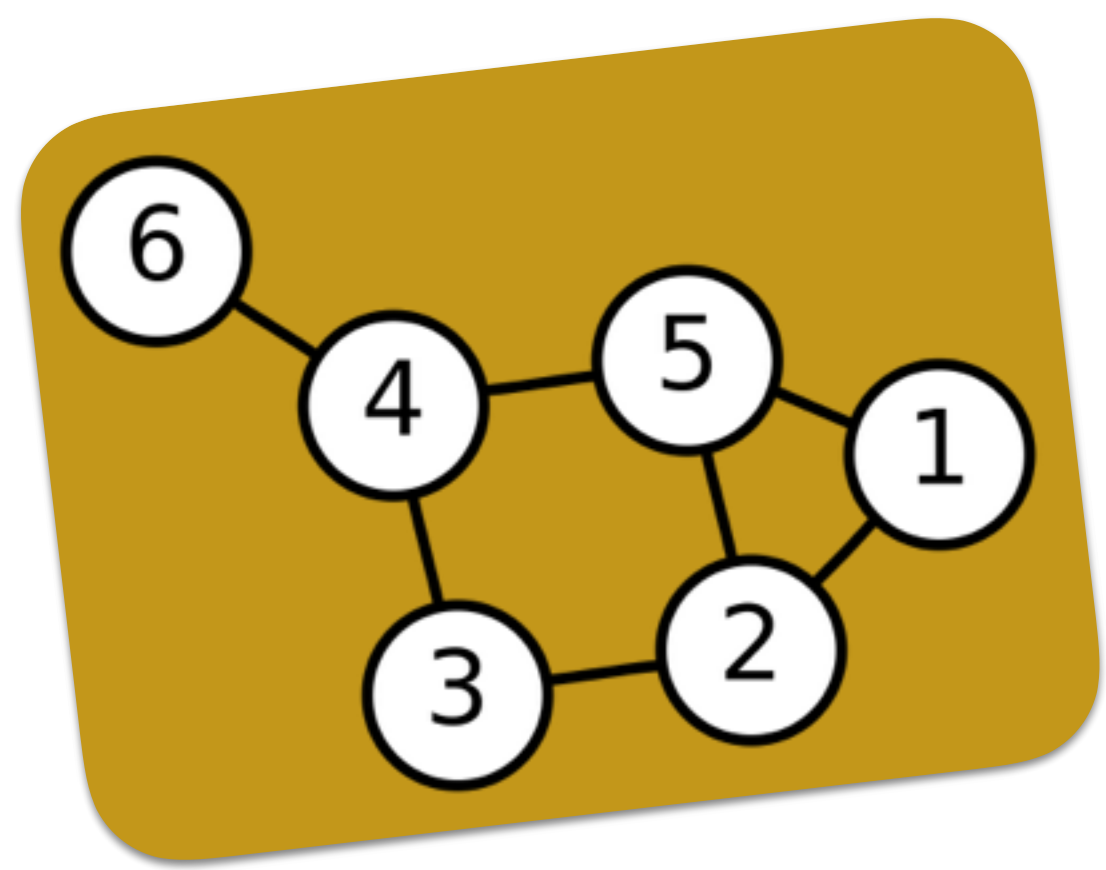
Graph Processing: hard to **distribute**

Same problems as parallelisation AND
deal with machine failures



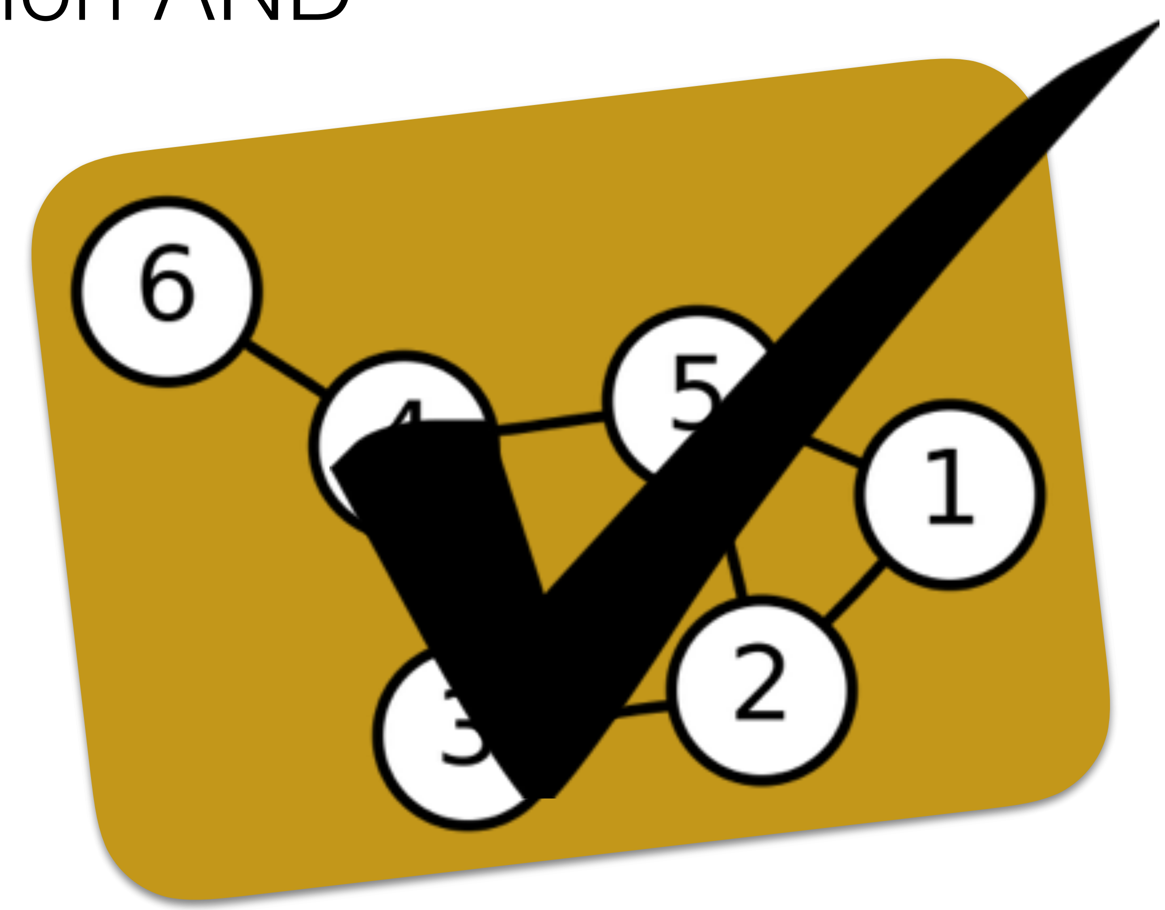
Graph Processing:
hard to **distribute**

Same problems as parallelisation AND
deal with machine failures



Graph Processing: hard to **distribute**

Same problems as parallelisation AND deal with machine failures



Two Birds

Two Birds

race conditions

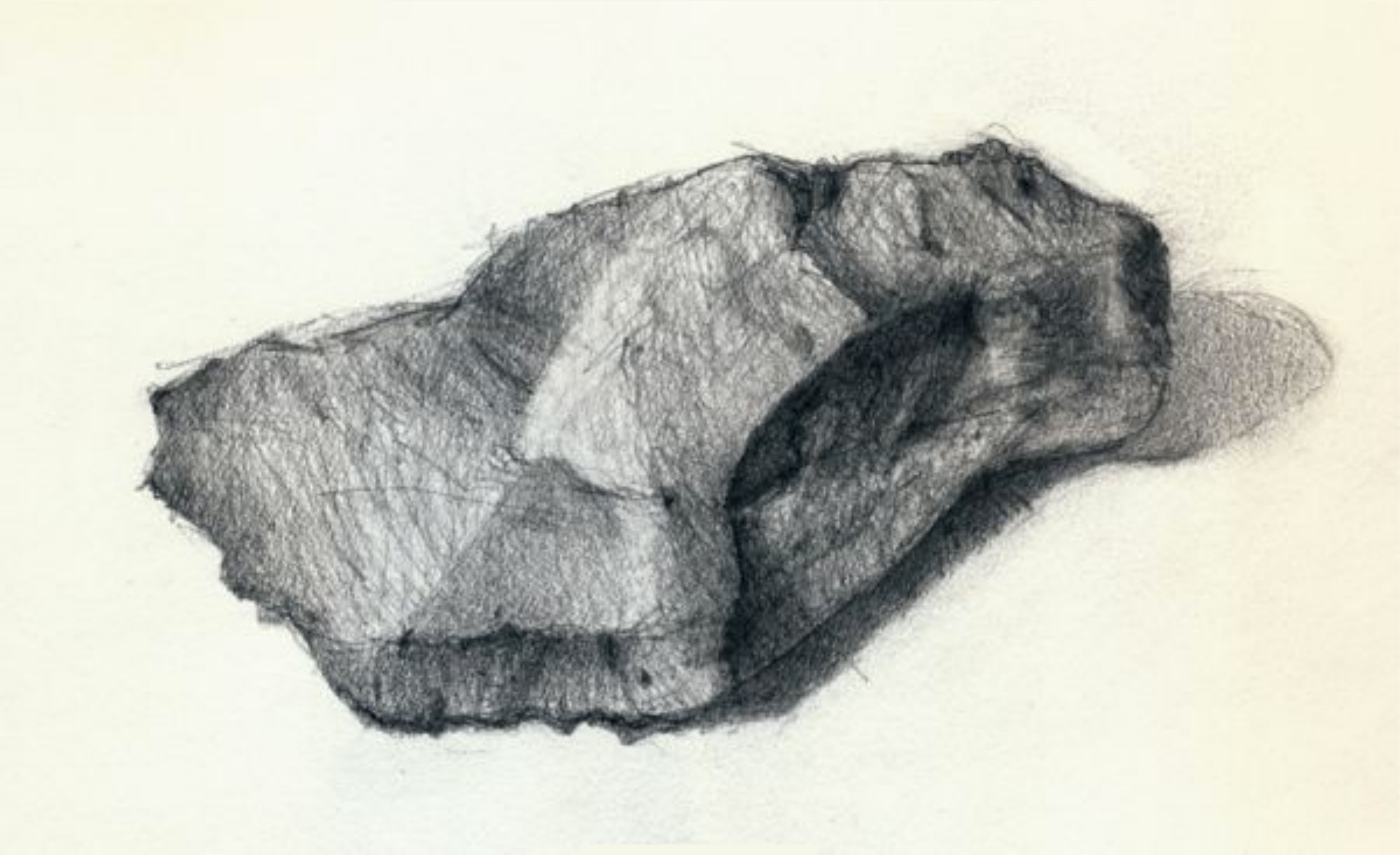
double espresso

Two Birds

race conditions

fault tolerance

One Stone



Pregel

One Stone

A programming abstraction/library

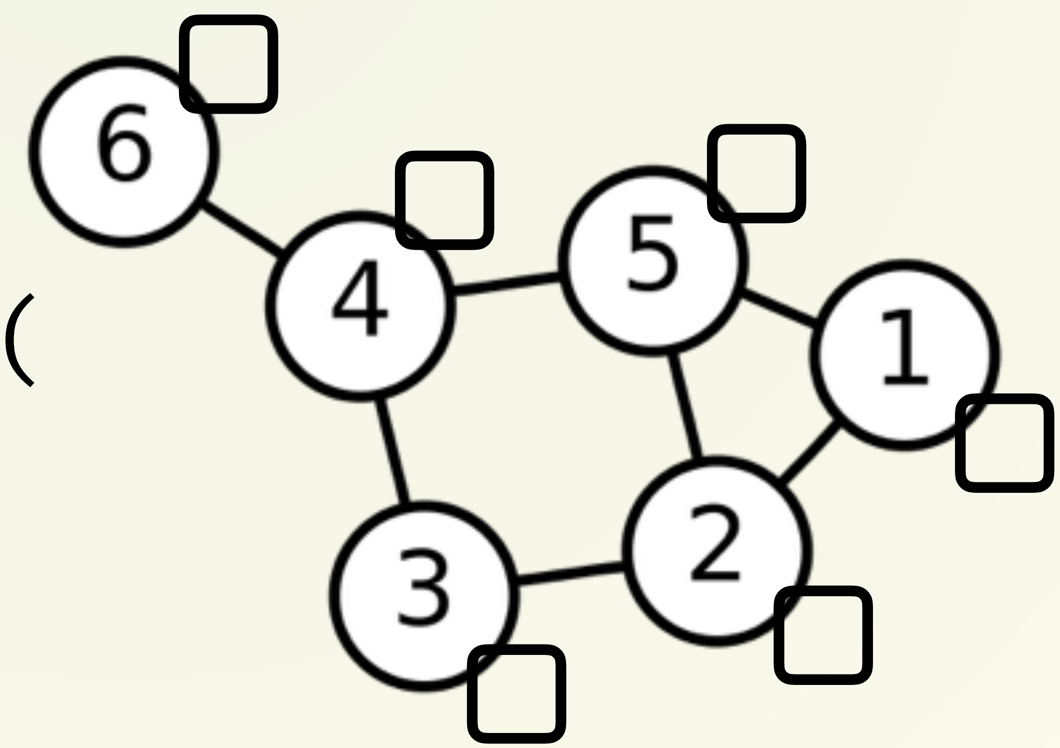
- actor based, with only local state
- message passing only

- actor based, with only local state
- message passing only

Example: Graph connectedness:

- One message: `PING`
- local state: a bool (inital: □ = `false`)

`assert(is_connected(`

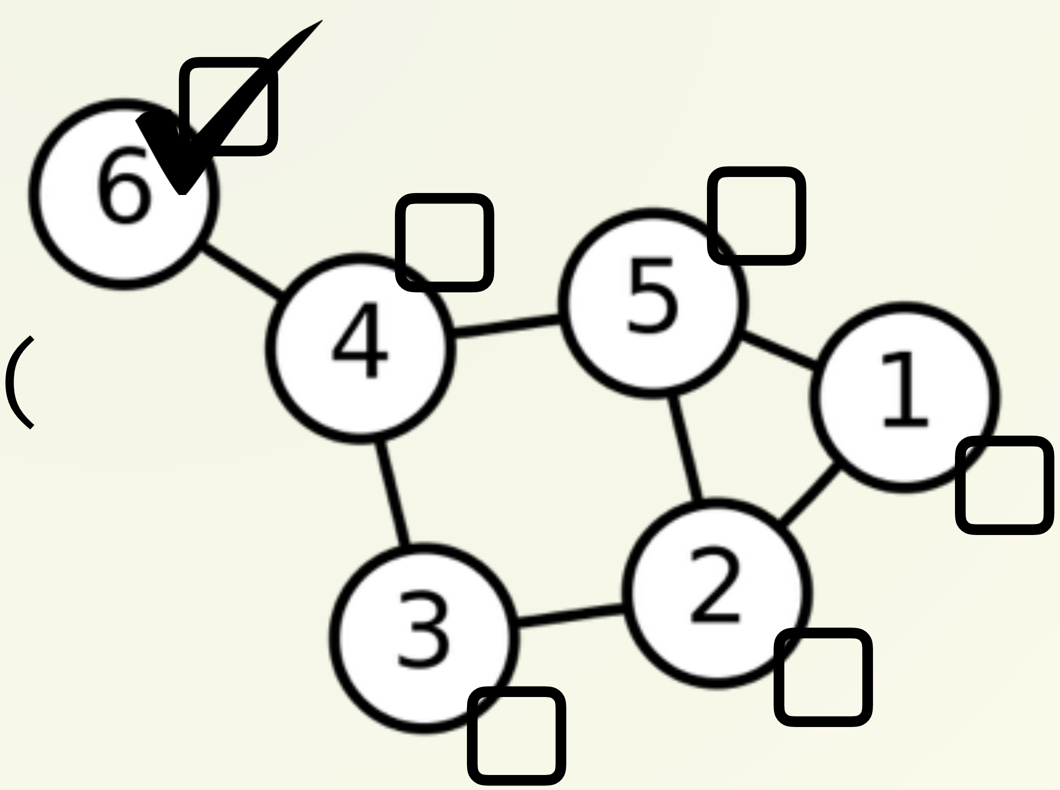


`))`

- actor based, with only local state
- message passing only

Example: Graph connectedness:

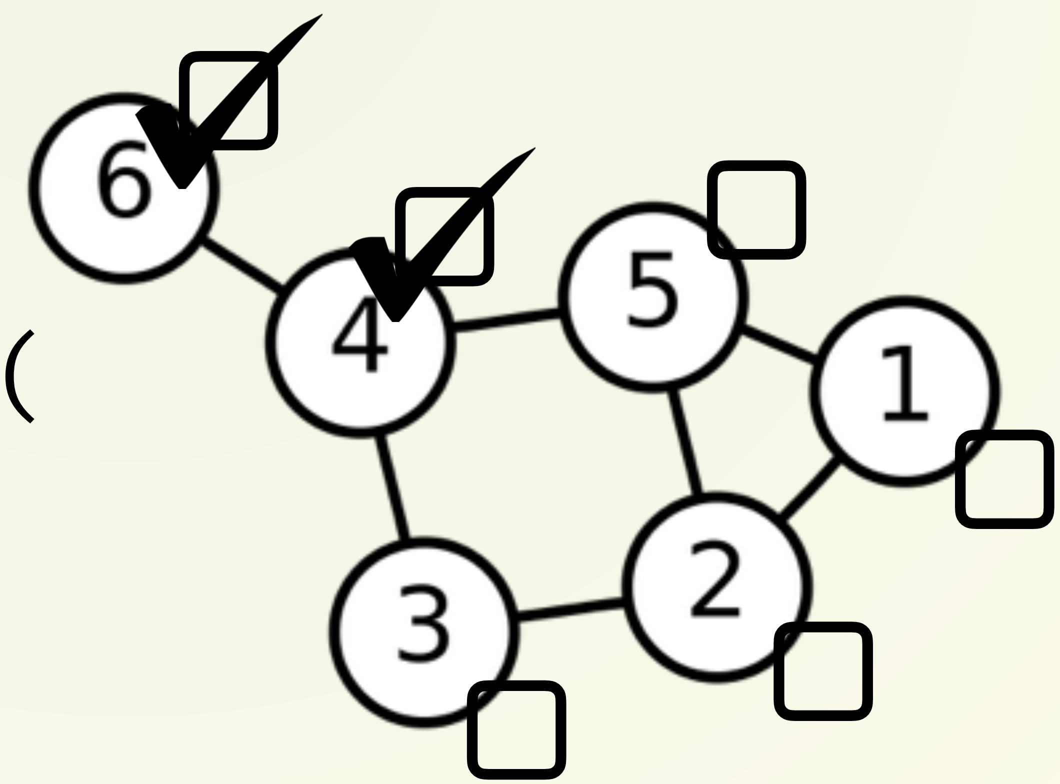
- One message: `PING`
- local state: a bool (inital: ☐ = `false`)

`assert(is_connected(`  `))`

- actor based, with only local state
- message passing only

Example: Graph connectedness:

- One message: `PING`
- local state: a bool (inital: ☐ = `false`)

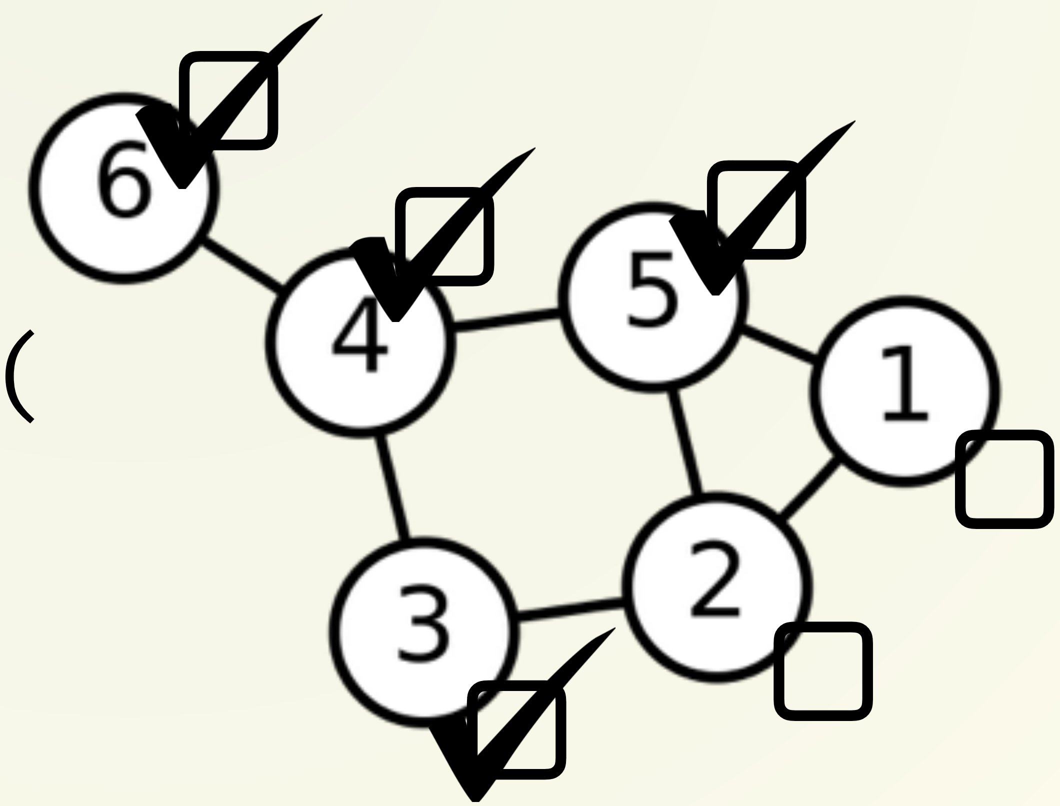
`assert(is_connected(`  `))`

- actor based, with only local state
- message passing only

Example: Graph connectedness:

- One message: `PING`
- local state: a bool (inital: ☐ = `false`)

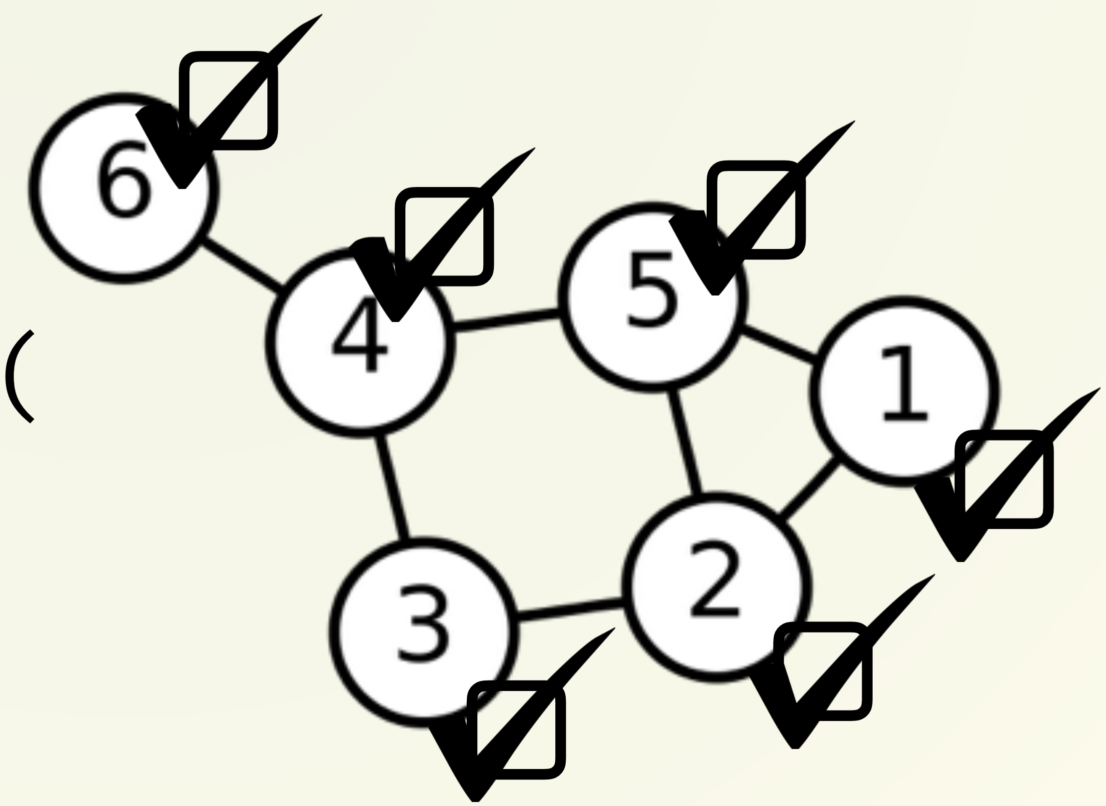
`assert(is_connected(`



`))`

- actor based, with only local state
- message passing only

Example: Graph connectedness:
- One message: `PING`
- local state: a bool (inital: ☐ = `false`)

`assert(is_connected(`  `))`

- actor based, with only local state
- message passing only

Example: Graph connectedness:
- One message: `PING`
- local state: a bool (inital: ☐ = `false`)

`assert(` ☑ `)`

- actor based, with only local state

⇒ **All state changes** have to go through the abstraction.

```
this.f.change()  /  send(CHANGE, this.f)
```

⇒ Can choose **when/how often to apply** state changes.

- message passing only

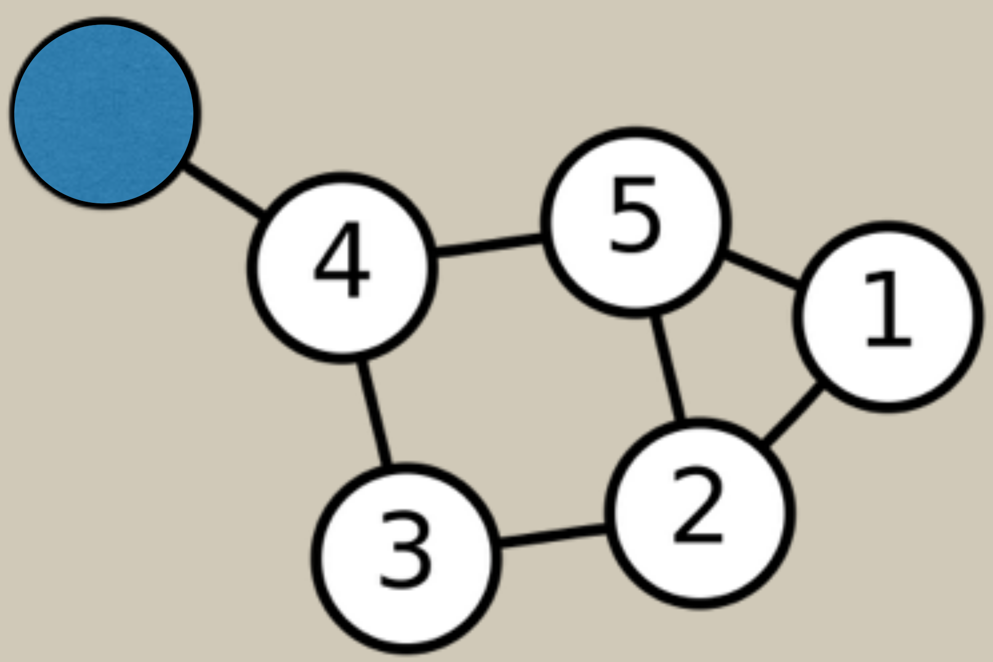
⇒ **All state changes** have to go through the abstraction.

⇒ Can choose **when/how often to apply** state changes.

That's all we need! :-)
We deliver all messages in an arbitrary, but deterministic order!

race conditions

Step 1



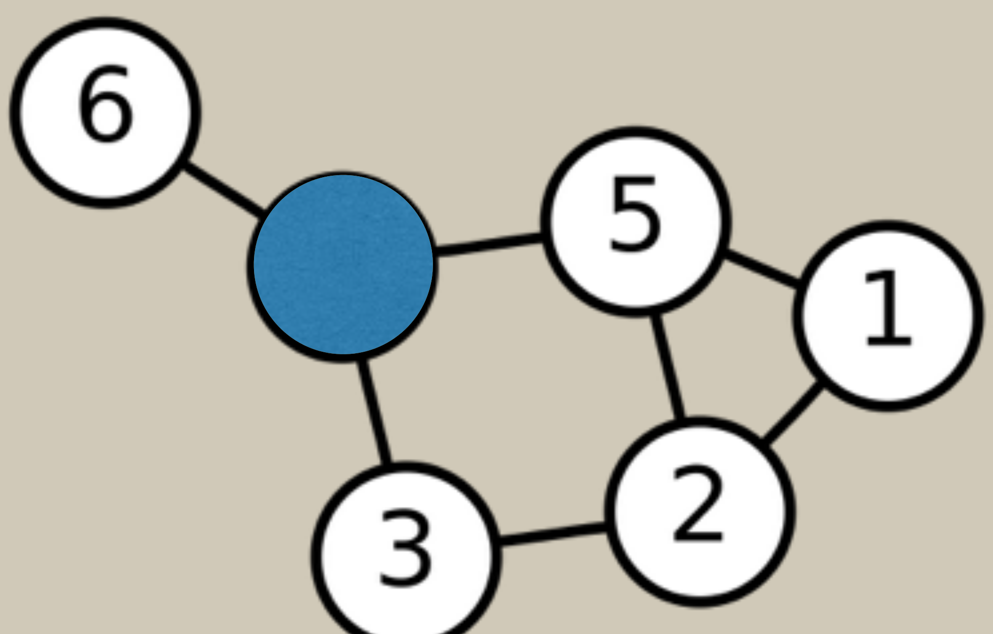
⇒ **All state changes** have to go through the abstraction.

⇒ Can choose **when/how often to apply** state changes.

That's all we need! :-)
We deliver all messages in an arbitrary, but deterministic order!

race conditions

Step 2



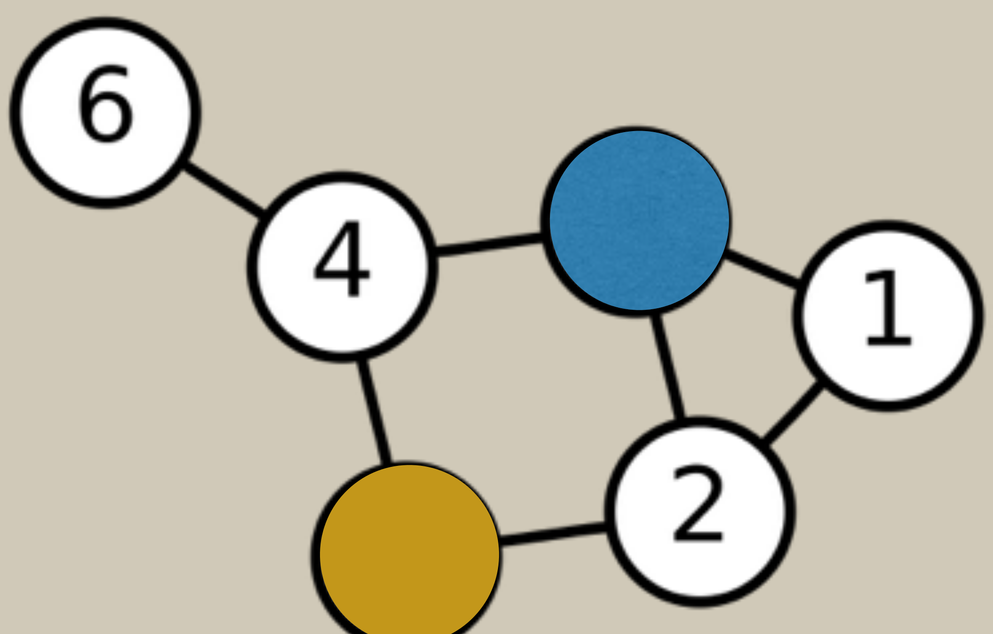
⇒ **All state changes** have to go through the abstraction.

⇒ Can choose **when/how often to apply** state changes.

That's all we need! :-)
We deliver all messages in an arbitrary, but deterministic order!

race conditions

Step 3



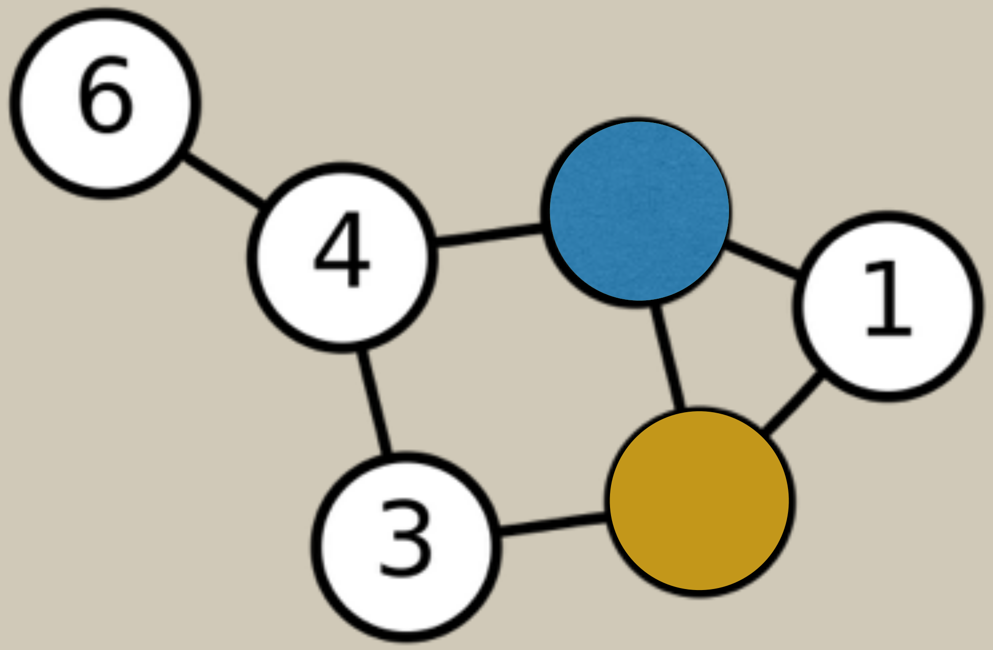
⇒ **All state changes** have to go through the abstraction.

⇒ Can choose **when/how often to apply** state changes.

That's all we need! :-)
super steps, and deliver messages in **deterministic order**!

race conditions

Step 4



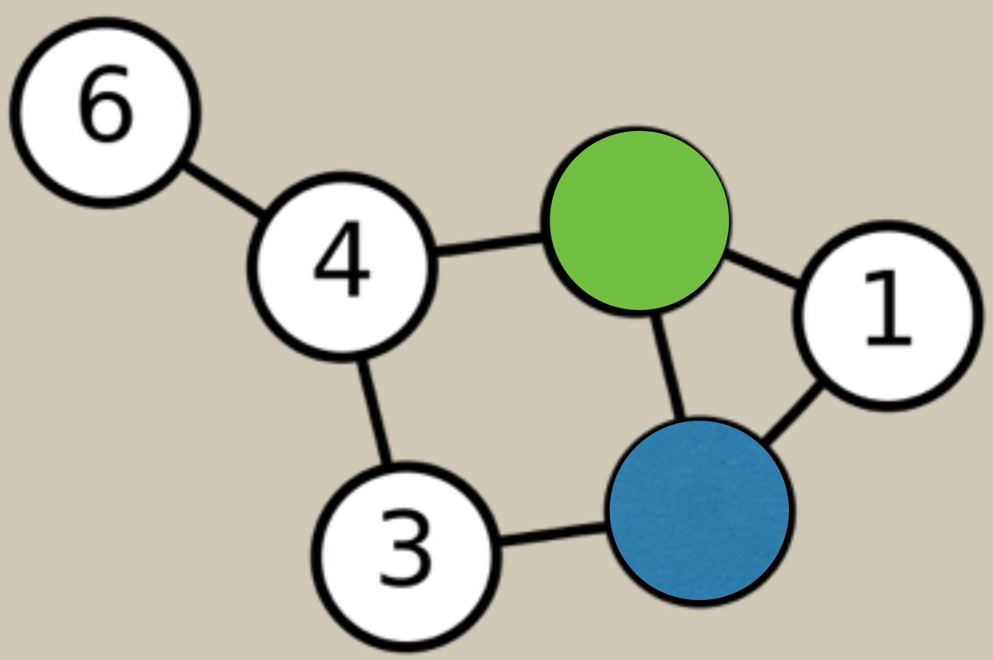
⇒ **All state changes** have to go through the abstraction.

⇒ Can choose **when/how often to apply** state changes.

That's all we need! :-)
super steps, and deliver messages in **deterministic order**!

race conditions

Step 4



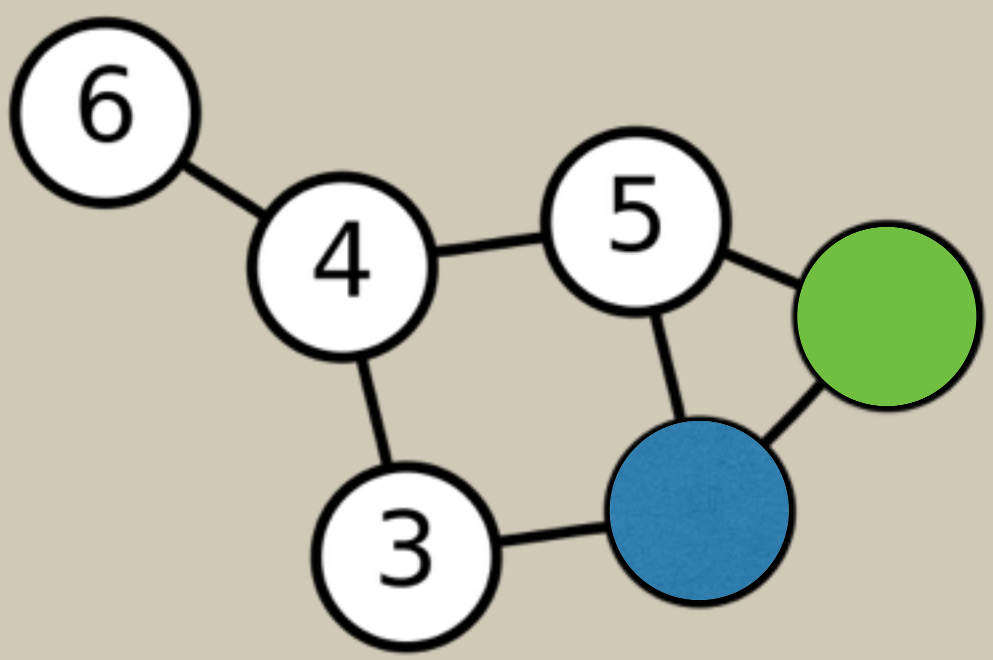
⇒ **All state changes** have to go through the abstraction.

⇒ Can choose **when/how often to apply** state changes.

That's all we need! :-)
super steps, and deliver
messages in **deterministic order**!

race conditions

Step 4



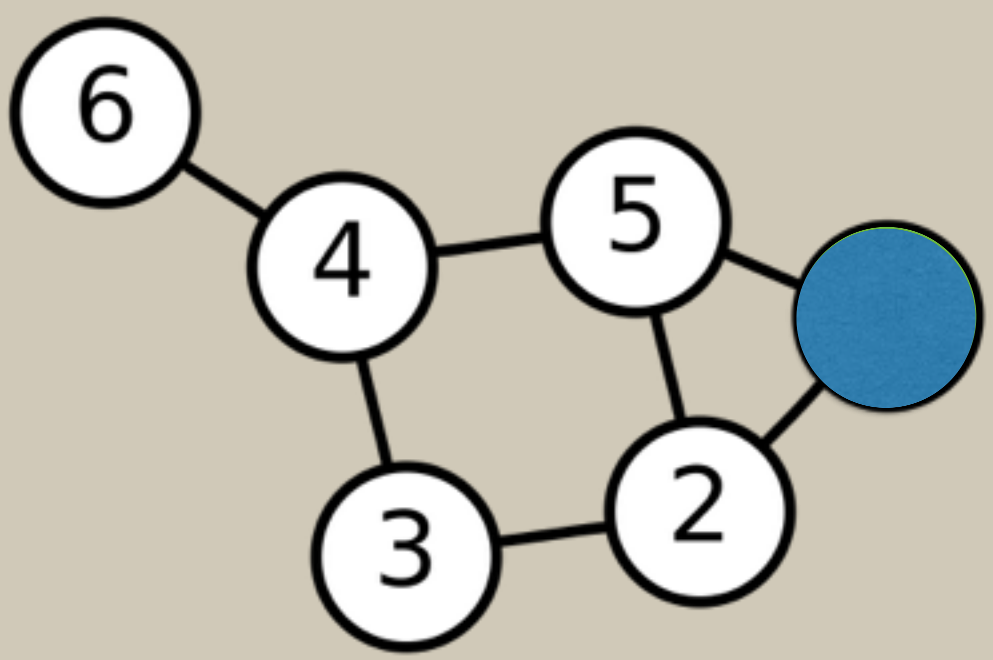
⇒ **All state changes** have to go through the abstraction.

⇒ Can choose **when/how often to apply** state changes.

That's all we need! :-)
super steps, and deliver
messages in **deterministic order**!

race conditions

Step 5



⇒ **All state changes** have to go through the abstraction.

⇒ Can choose **when/how often to apply** state changes.

That's all we need! :-)
super steps, and deliver messages in **deterministic order**!

race conditions

⇒ **All state changes** have to go through the abstraction.

⇒ Can choose **when/how often to apply** state changes.

That's all we need! :-)
super steps, and deliver
messages in **deterministic order**!

⇒ **All state changes** have to go through the abstraction.

⇒ Can choose **when/how often to apply** state changes.

That's all we need! :-)
super steps, and persist machine state before each step

"Persistent data is stored as files on a distributed storage system, GFS [19], or in Bigtable [9], and temporary data such as buffered messages on local disk."

Invariant: only successful states are persisted

Safety for performance tradeoff.

⇒ **All state changes** have to go through the abstraction.

⇒ Can choose **when/how often to apply** state changes.

That's all we need! :-)
super steps, and persist machine state before each step

fault tolerance

"Persistent data is stored as files on a distributed storage system, GFS [19], or in Bigtable [9], and temporary data such as buffered messages on local disk."

Invariant: only successful states are persisted

Safety for performance tradeoff.

Slides at: http://stbr.me/pregel-presentation