MapReduce

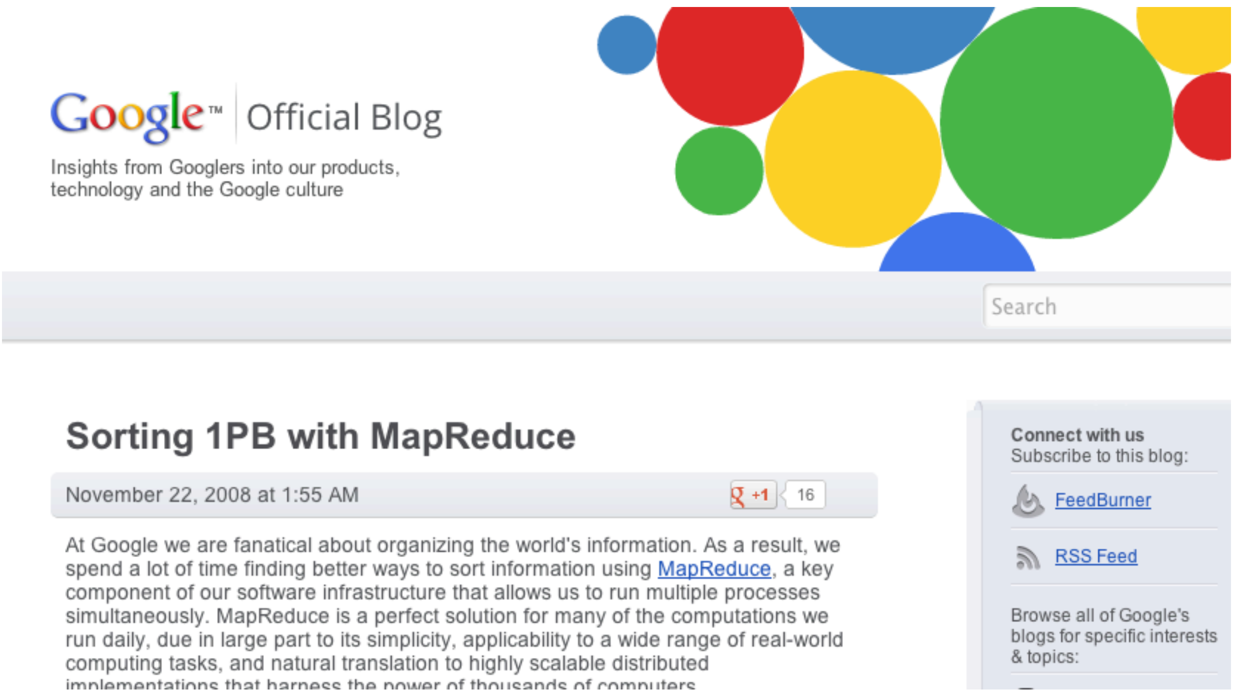
What it is, and why it is so popular

Luigi Laura

Dipartimento di Informatica e Sistemistica
"Sapienza" Università di Roma

Rome, May 17^{th}, 2012

Motivations: sorting one **Peta**byte



Motivations: sorting...

- Nov. 2008: 1TB, 1000 computers, 68 seconds.
  Previous record was 910 computers, 209 seconds.
- Nov. 2008: 1PB, 4000 computers, 6 hours; 48k harddisks...
- Sept. 2011: 1PB, 8000 computers, 33 minutes.
- Sept. 2011: 10PB, 8000 computers, 6 hours and 27 minutes.

The last slide of this talk...

"The beauty of MapReduce is that any programmer can understand it, and its power comes from being able to harness thousands of computers behind that simple interface"

David Patterson

Outline of this talk

What is MapReduce?

MapReduce is a distributed computing paradigm that's here now

- Designed for 10,000+ node clusters
- Very popular for processing large datasets
- Processing over 20 petabytes per day [Google, Jan 2008]
- But virtually NO analysis of MapReduce algorithms

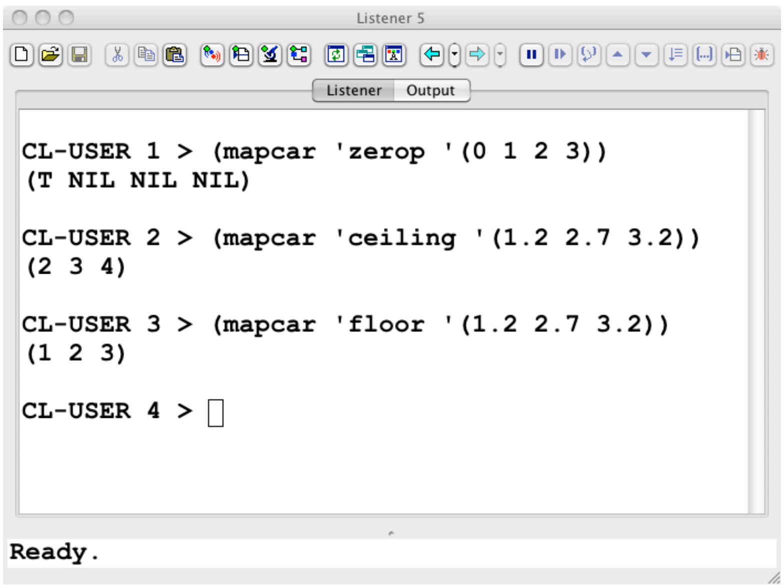
The origins...

"Our abstraction is inspired by the **map** *and* **reduce** *primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a* **map** *operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a* **reduce** *operation to all the values that shared the same key, in order to combine the derived data appropriately."*

Jeffrey Dean and Sanjay Ghemawat [OSDI 2004]

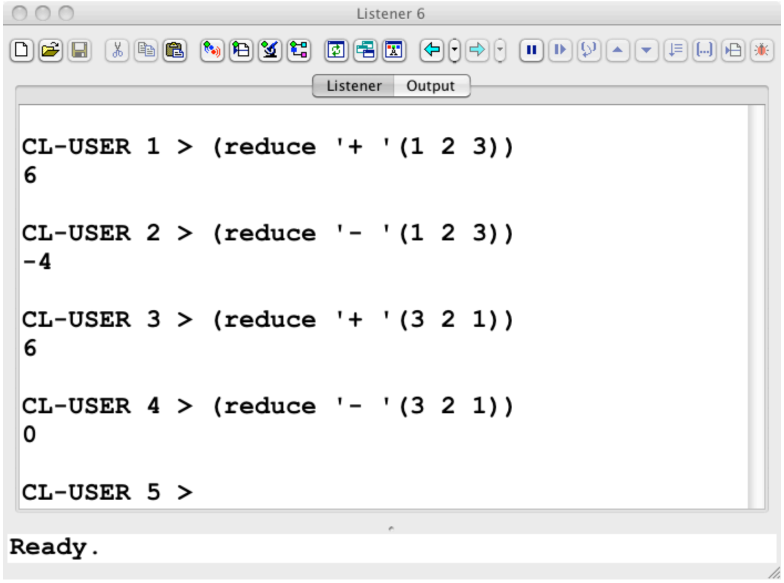
Map in Lisp

The *map(car)* is a function that calls its first argument with each element of its second argument, in turn.



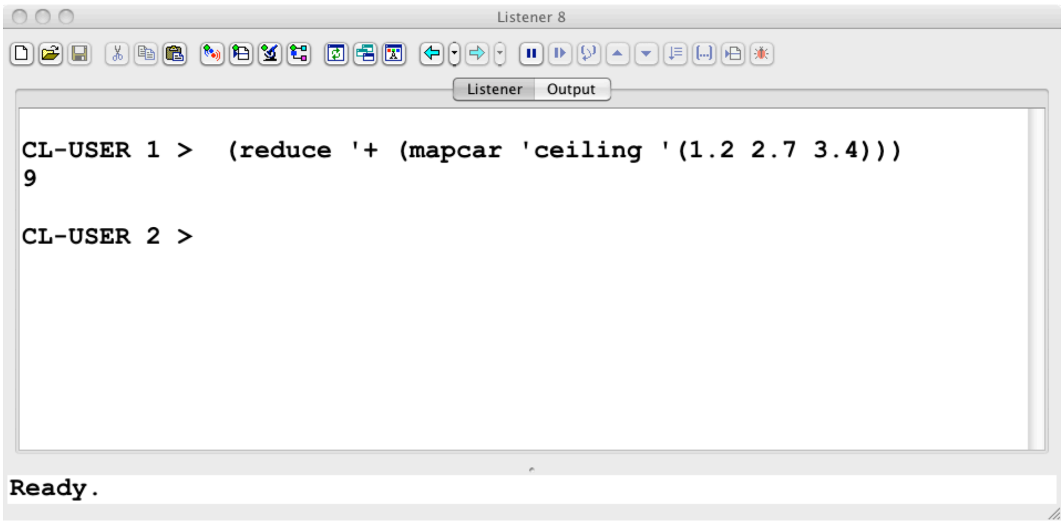
Reduce in Lisp

The *reduce* is a function that returns a single value constructed by calling the first argument (a function) function on the first two items of the second argument (a sequence), then on the result and the next item, and so on .



MapReduce in Lisp

Our first MapReduce program :-)

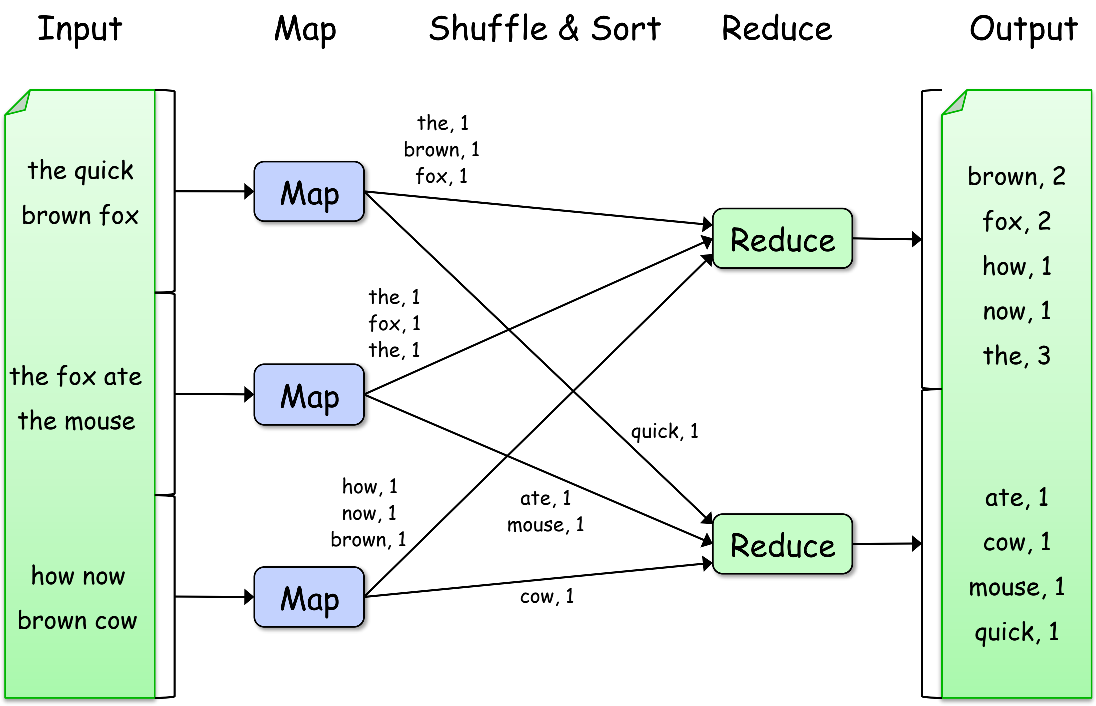


THE example in MapReduce: Word Count

```
def mapper(line):
    foreach word in line.split():
        output(word, 1)

def reducer(key, values):
    output(key, sum(values))
```

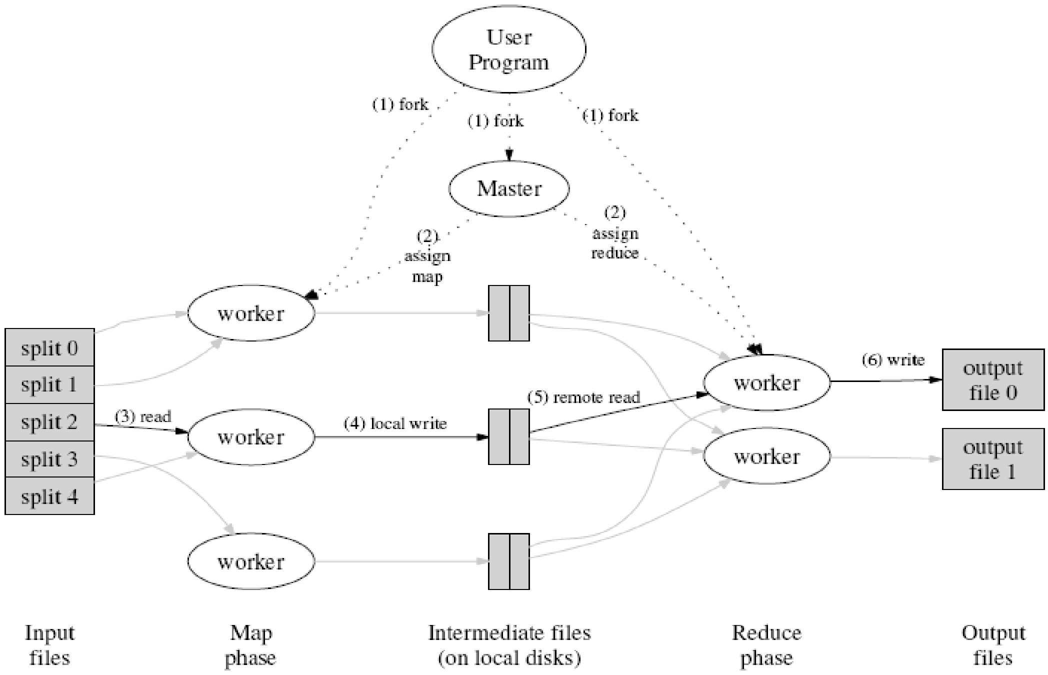
Word Count Execution



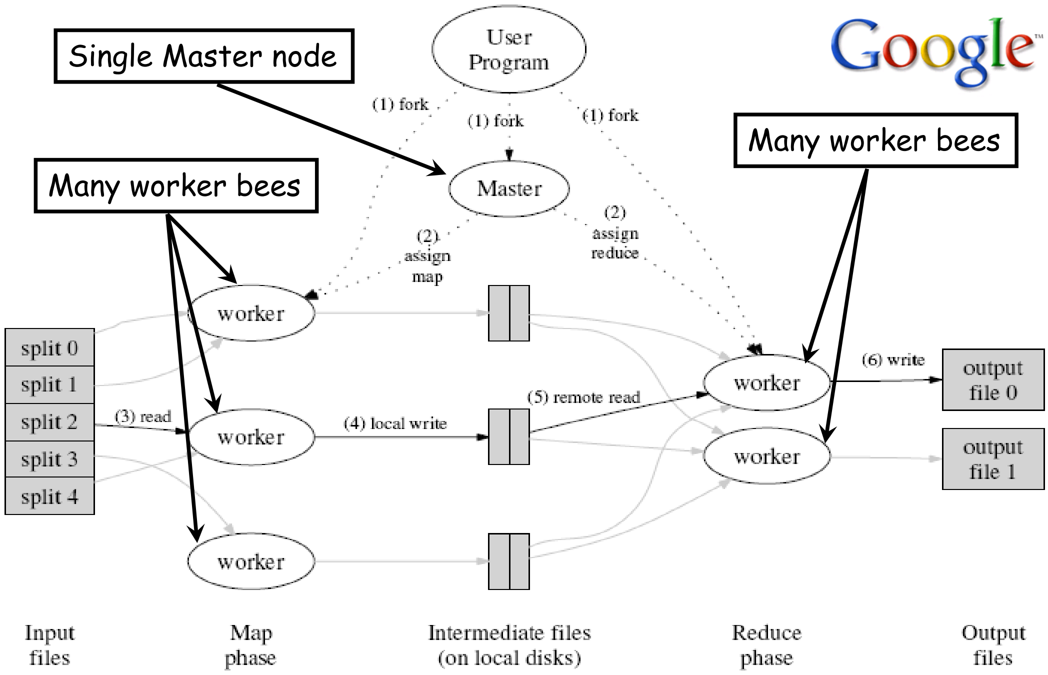
MapReduce Execution Details

- Single master controls job execution on multiple slaves
- Mappers preferentially placed on same node or same rack as their input block
  - Minimizes network usage
- Mappers save outputs to local disk before serving them to reducers
  - Allows recovery if a reducer crashes
  - Allows having more reducers than nodes

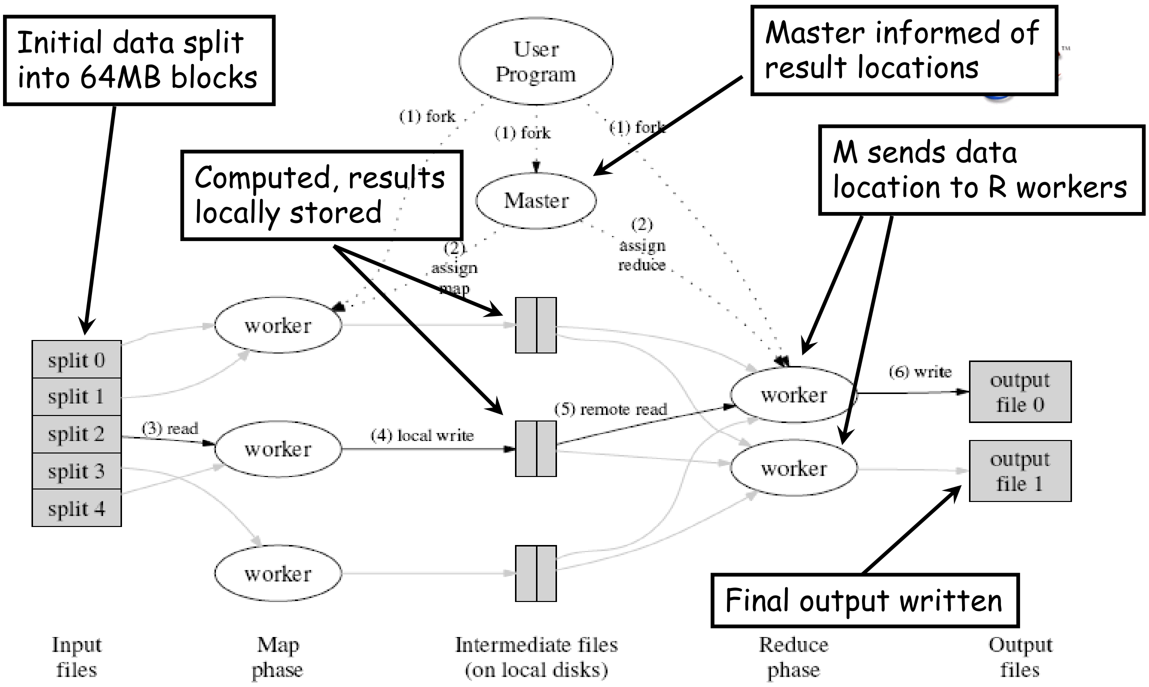
MapReduce Execution Details



MapReduce Execution Details



MapReduce Execution Details



MapReduce Execution Details

Process	Time ---------------------->			
User Program	MapReduce()	... wait ...		
Master	Assign tasks to worker machines...			
Worker 1	Map 1	Map 3		
Worker 2	Map 2			
Worker 3	Read 1.1	Read 1.3	Read 1.2	Reduce 1
Worker 4	Read 2.1	Read 2.2	Read 2.3	Reduce 2

Exercise!

Word Count is trivial...

how do we compute SSSP in MapReduce?

Hint: we do not need our algorithm to be feasible...
just a proof of concept!

Programming Model

- MapReduce library is **extremely** easy to use
- Involves setting up only a few parameters, and defining the map() and reduce() functions
  - Define map() and reduce()
  - Define and set parameters for MapReduceInput object
  - Define and set parameters for MapReduceOutput object
  - Main program

Most important/unknown/hidden feature: if a single key combined mappers output is too large for a single reducer, then it is handled "as a tournament" between several reducers!

What is MapReduce/Hadoop used for?

- At Google:
  - Index construction for Google Search
  - Article clustering for Google News
  - Statistical machine translation
- At Yahoo!:
  - "Web map" powering Yahoo! Search
  - Spam detection for Yahoo! Mail
- At Facebook:
  - Data mining
  - Ad optimization
  - Spam detection

Large Scale PDF generation - The Problem

- The New York Times needed to generate PDF files for 11,000,000 articles (every article from 1851-1980) in the form of images scanned from the original paper
- Each article is composed of numerous TIFF images which are scaled and glued together
- Code for generating a PDF is relatively straightforward

Large Scale PDF generation - Technologies Used

- Amazon Simple Storage Service (S3) [0.15$/GB/month]
  - Scalable, inexpensive internet storage which can store and retrieve any amount of data at any time from anywhere on the web
  - Asynchronous, decentralized system which aims to reduce scaling bottlenecks and single points of failure
- Hadoop running on Amazon Elastic Compute Cloud (EC2) [0.10$/hour]
  - Virtualized computing environment designed for use with other Amazon services (especially S3)

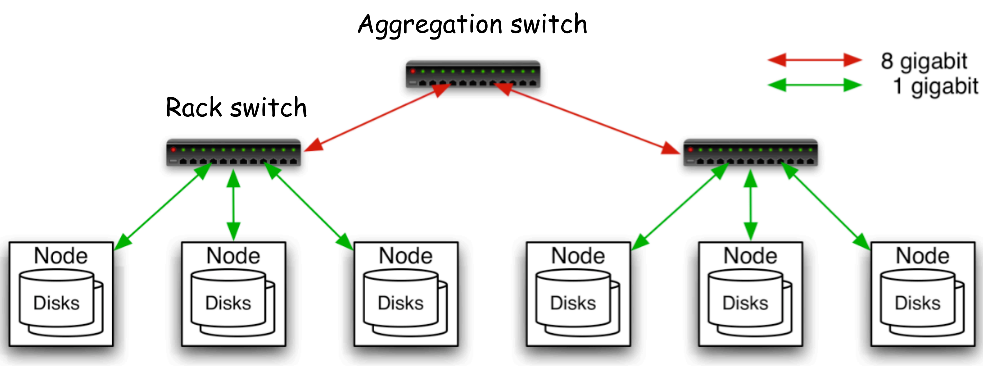
Large Scale PDF generation - Results

- 4TB of scanned articles were sent to S3
- A cluster of EC2 machines was configured to distribute the PDF generation via Hadoop
- Using 100 EC2 instances and 24 hours, the New York Times was able to convert 4TB of scanned articles to 1.5TB of PDF documents

Hadoop

- MapReduce is a working framework used **inside** Google.
- Apache Hadoop is a top-level Apache project being built and used by a global community of contributors, using the Java programming language.
- Yahoo! has been the largest contributor

Typical Hadoop Cluster



- 40 nodes/rack, 1000-4000 nodes in cluster
- 1 Gbps bandwidth within rack, 8 Gbps out of rack
- Node specs (Yahoo terasort): 8 x 2GHz cores, 8 GB RAM, 4 disks (= 4 TB?)

Typical Hadoop Cluster

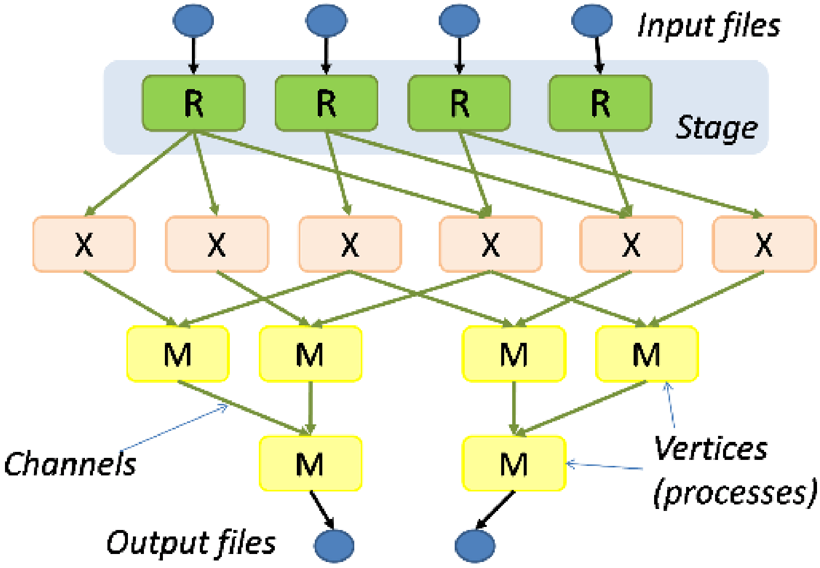
Hadoop Demo

- Now we see Hadoop in action...
- ...as an example, we consider the Fantacalcio computation...
- ... code and details available from:
  `https://github.com/bernarpa/FantaHadoop`

Microsoft Dryad

- A Dryad programmer writes several sequential programs and connects them using one-way channels.
- The computation is structured as a directed graph: programs are graph vertices, while the channels are graph edges.
- A Dryad job is a graph generator which can synthesize any directed acyclic graph.
- These graphs can even change during execution, in response to important events in the computation.

Microsoft Dryad - A job



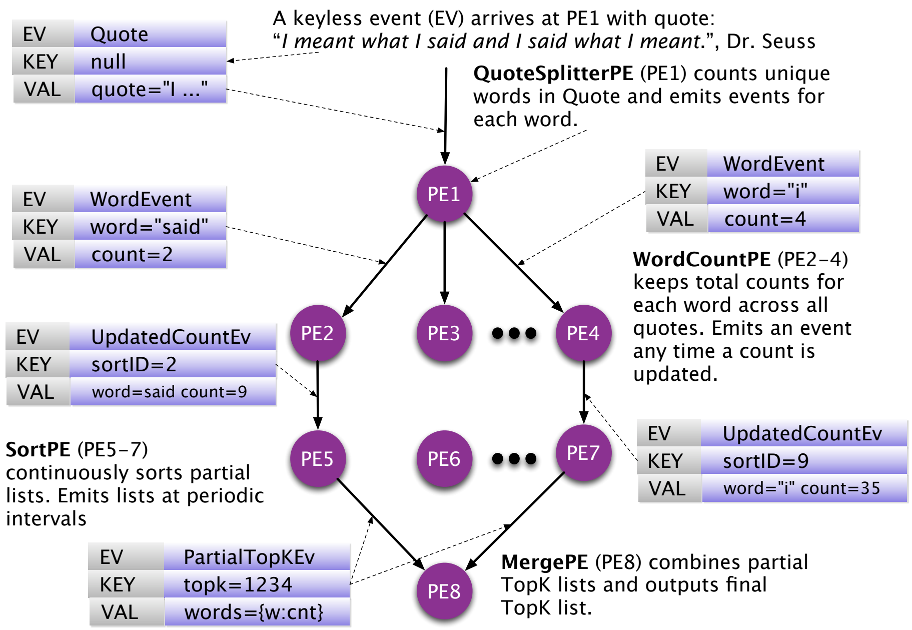
Yahoo! S4: Distributed Streaming Computing Platform

S4 is a general-purpose, distributed, scalable, partially fault-tolerant, pluggable platform that allows programmers to easily develop applications for processing **continuous unbounded streams of data.**

Keyed data events are routed with affinity to Processing Elements (PEs), which consume the events and do one or both of the following:

- emit one or more events which may be consumed by other PEs,
- publish results.

Yahoo! S4 - Word Count example



PE ID	PE Name	Key Tuple
PE1	QuoteSplitterPE	null
PE2	WordCountPE	word="said"
PE4	WordCountPE	word="i"
PE5	SortPE	sortID=2
PE7	SortPE	sortID=9
PE8	MergePE	topK=1234

Google Pregel: a System for Large-Scale Graph Processing

- Vertex-centric approach
- Message passing to neighbours
- *Think like a vertex* mode of programming

PageRank example!

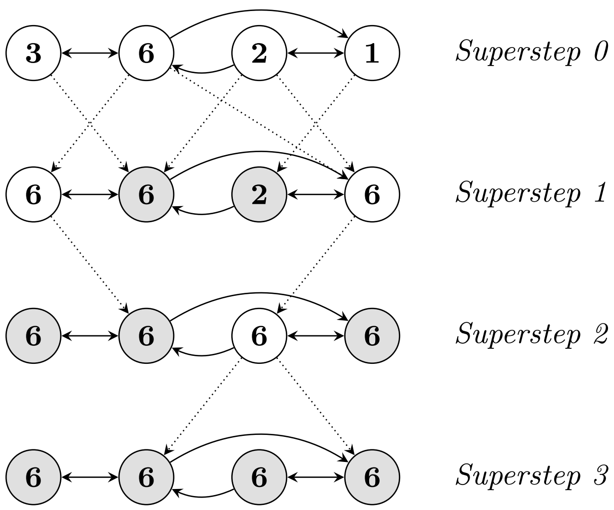
Google Pregel

Pregel computations consist of a sequence of iterations, called supersteps. During a superstep the framework invokes a user-defined function for each vertex, conceptually in parallel. The function specifies behavior at a single vertex V and a single superstep S. It can:

- read messages sent to V in superstep $S-1$,
- send messages to other vertices that will be received at superstep $S+1$, and
- modify the state of V and its outgoing edges.

Messages are typically sent along outgoing edges, but a message may be sent to any vertex whose identifier is known.

Google Pregel



Maximum Value Example

Twitter Storm

"Storm makes it easy to write and scale complex realtime computations on a cluster of computers, doing for realtime processing what Hadoop did for batch processing. Storm guarantees that every message will be processed. And it's fast — you can process millions of messages per second with a small cluster. Best of all, you can write Storm topologies using any programming language."

Nathan Marz

Twitter Storm: features

- Simple programming model. Similar to how MapReduce lowers the complexity of doing parallel batch processing, Storm lowers the complexity for doing real-time processing.
- Runs any programming language. You can use any programming language on top of Storm. Clojure, Java, Ruby, Python are supported by default. Support for other languages can be added by implementing a simple Storm communication protocol.
- Fault-tolerant. Storm manages worker processes and node failures. Horizontally scalable. Computations are done in parallel using multiple threads, processes and servers.
- Guaranteed message processing. Storm guarantees that each message will be fully processed at least once. It takes care of replaying messages from the source when a task fails.
- Local mode. Storm has a "local mode" where it simulates a Storm cluster completely in-process. This lets you develop and unit test topologies quickly.

Theoretical Models

So far, two models:

- Massive Unordered Distributed (MUD) Computation, by Feldman, Muthukrishnan, Sidiropoulos, Stein, and Svitkina [SODA 2008]
- A Model of Computation for MapReduce (MRC), by Karloff, Suri, and Vassilvitskii [SODA 2010]

Massive Unordered Distributed (MUD)

An algorithm for this platform consist of three functions:

- a local function to take a single input data item and output a message,
- an aggregation function to combine pairs of messages, and in some cases
- a final postprocessing step

More formally, a MUD algorithm is a triple $m = (\Phi, \oplus, \eta)$:

- $\Phi : \Sigma \to Q$ maps an input item Σ to a message Q.
- $\oplus : Q \times Q \to Q$ maps two messages to a single one.
- $\eta : Q \to \Sigma$ produces the final output.

Massive Unordered Distributed (MUD) - The results

- Any deterministic streaming algorithm that computes a symmetric function $\Sigma^n \to \Sigma$ can be simulated by a mud algorithm with the same communication complexity, and the square of its space complexity.
- This result generalizes to certain approximation algorithms, and randomized algorithms with public randomness (i.e., when all machines have access to the same random tape).

Massive Unordered Distributed (MUD) - The results

- The previous claim does not extend to richer symmetric function classes, such as when the function comes with a *promise* that the domain is guaranteed to satisfy some property (e.g., finding the diameter of a graph known to be connected), or the function is *indeterminate*, that is, one of many possible outputs is allowed for "successful computation" (e.g., finding a number in the highest 10% of a set of numbers). Likewise, with private randomness, the preceding claim is no longer true.

- The simulation takes time $\Omega(2^{polylog(n)})$ from the use of Savitch's theorem.
- Therefore the simulation is **not** a practical solution for executing streaming algorithms on distributed systems.

Map Reduce Class (MRC)

Three Guiding Principles
The input size is n

Space Bounded memory per machine

- Cannot fit all of input onto one machine
- Memory per machine $n^{1-\varepsilon}$

Time Small number of rounds

- Strive for constant, but OK with $\log_{O(1)} n$
- Polynomial time per machine (No streaming constraints)

Machines Bounded number of machines

- Substantially sublinear number of machines
- Total $n^{1-\varepsilon}$

MRC & NC

Theorem: Any NC algorithm using at most $n^{2-\varepsilon}$ processors and at most $n^{2-\varepsilon}$ memory can be simulated in MRC.

Instant computational results for MRC:

- Matrix inversion [Csanky's Algorithm]
- Matrix Multiplication & APSP
- Topologically sorting a (dense) graph
- ...

But the simulation does not exploit full power of MR

- Each reducer can do sequential computation

Open Problems

- Both the models seen are not a model, in the sense that we cannot compare algorithms.
- **We need such a model!**
- Both the reductions seen are useful only from a theoretical point of view, i.e. we cannot use them to convert streaming/NC algorithms into MUD/MRC ones.
- **We need to keep on designing algorithms the old fashioned way!!**

Things I (almost!) did not mention

In this overview several details[1] are not covered:

- Google File System (GFS), used by MapReduce
- Hadoop Distributed File System, used by Hadoop
- The Fault-tolerance of these and the other frameworks...
- ... algorithms in MapReduce (very few, so far...)

Outline: Graph Algorithms in MR?

Is there any memory efficient constant round algorithm for connected components in sparse graphs?

- Let us start from computation of MST of Large-Scale graphs
- Map Reduce programming paradigm
- *Semi-External* and *External* Approaches
- Work in Progress and Open Problems . . .

Notation Details

Given a weighted undirected graph $G = (V, E)$

- n is the number of vertices
- N is the number of edges
  (size of the input in many MapReduce works)
- all of the edge weights are unique
- G is connected

Sparse Graphs, Dense Graphs and Machine Memory I

(1) Semi-External MapReduce graph algorithm.
Working memory requirement of any map or reduce computation
$O(N^{1-\epsilon})$, for some $\epsilon > 0$

(2) External MapReduce graph algorithm.
Working memory requirement of any map or reduce computation
$O(n^{1-\epsilon})$, for some $\epsilon > 0$

Similar definitions for *streaming* and *external memory* graph algorithms

$\boxed{O(N) \text{ not allowed!}}$

Sparse Graphs, Dense Graphs and Machine Memory II

(1) G is *dense*, i.e., $N = n^{1+c}$

The design of a semi-external algorithm:

- makes sense for some $\frac{c}{1+c} \geq \epsilon > 0$
  (otherwise it is an external algorithm, $O(N^{1-\epsilon}) = O(n^{1-\epsilon})$)
- allows to store G vertices

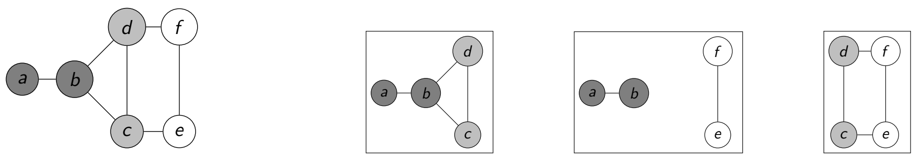
(2) G is *sparse*, i.e., $N = O(n)$

- no difference between semi-external and external algorithms
- storing G vertices is never allowed

Karloff et al. algorithm (SODA '10) I

mrmodelSODA10

(1) Map Step 1.
Given a number k, randomly partition the set of vertices into k equally sized subsets: $G_{i,j}$ is the subgraph given by $(V_i \cup V_j, E_{i,j})$.



G G_{12} G_{13} G_{23}

Karloff et al. algorithm (SODA '10) II

(2) Reduce Step 1.
For each of the $\binom{k}{2}$ subgraphs $G_{i,j}$, compute the MST (forest) $M_{i,j}$.

---

(3) Map Step 2.
Let H be the graph consisting of all of the edges present in some $M_{i,j}$: $H = (V, \bigcup_{i,j} M_{i,j})$: map H to a single reducer \$.

(4) Reduce Step 2.
Compute the MST of H.

Karloff et al. algorithm (SODA '10) III

The algorithm is *semi-external*, for dense graphs.

- if G is c-dense and if $k = n^{\frac{c'}{2}}$, for some $c \geq c' > 0$: with high probability, the memory requirement of any map or reduce computation is

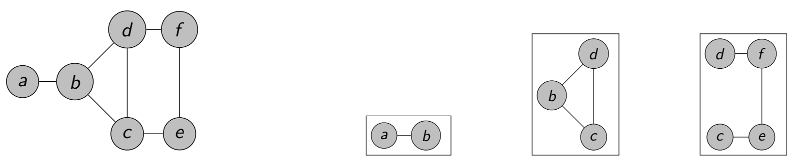
$$O(N^{1-\epsilon}) \tag{1}$$

- it works in $2 = O(1)$ rounds

Lattanzi et al. algorithm (SPAA '11) I

filteringSPAA11

(1) MAP STEP i.
Given a number k, randomly partition the set of edges into $\frac{|E|}{k}$ equally sized subsets: G_i is the subgraph given by (V_i, E_i)



G $\quad G_1 \quad G_2 \quad G_3$

Lattanzi et al. algorithm (SPAA '11) II

(2) REDUCE STEP i.
For each of the $\frac{|E|}{k}$ subgraphs G_i, computes the graph G_i', obtained by removing from G_i any edge that is guaranteed not to be a part of any MST because it is the heaviest edge on some cycle in G_i.

Let H be the graph consisting of all of the edges present in some G_i'

- if $|E| \leq k \rightarrow$ the algorithm ends
  (H is the MST of the input graph G)
- otherwise $\rightarrow$ start a new round with H as input

Lattanzi et al. algorithm (SPAA '11) III

The algorithm is *semi-external*, for dense graphs.

- if G is c-dense and if $k = n^{1+c'}$, for some $c \geq c' > 0$:
  the memory requirement of any map or reduce computation is

$$O(n^{1+c'}) = O(N^{1-\epsilon}) \qquad (2)$$

  for some

$$\frac{c'}{1+c'} \geq \epsilon > 0 \qquad (3)$$

- it works in $\lceil \frac{c}{c'} \rceil = O(1)$ rounds

Summary

	[mrmodelSODA10]	[filteringSPAA11]
	G is c-dense, and $c \geq c' > 0$	
	if $k = n^{\frac{c'}{2}}$, whp	if $k = n^{1+c'}$
Memory	$O(N^{1-\epsilon})$	$O(n^{1+c'}) = O(N^{1-\epsilon})$
Rounds	2	$\lceil \frac{c}{c'} \rceil = O(1)$

Table: Space and Time complexity of algorithms discussed so far.

Experimental Settings (thanks to A. Paolacci)

- **Data Set.**
  Web Graphs, from hundreds of thousand to 7 millions vertices
  `http://webgraph.dsi.unimi.it/`
- **Map Reduce framework.**
  Hadoop 0.20.2 (pseudo-distributed mode)
- **Machine.**
  CPU Intel i3-370M (3M cache, 2.40 Ghz), RAM 4GB, Ubuntu Linux.
- **Time Measures.**
  Average of 10 rounds of the algorithm on the same instance

Preliminary Experimental Evaluation I

Memory Requirement in [**mrmodelSODA10**]

	Mb	c	n^{1+c}	$k = n^{1+c'}$	round 1[1]	round 2[1]
cnr-2000	43.4	0.18	3.14	3	7.83	4.82
in-2004	233.3	0.18	3.58	3	50.65	21.84
indochina-2004	2800	0.21	5.26	5	386.25	126.17

Using smaller values of k (decreasing parallelism)

- *decreases* round 1 output size $\rightarrow$ round 2 time $\ddot\smile$
- *increases* memory and time requirement of round 1 reduce step $\ddot\frown$

[1] output size in Mb

Preliminary Experimental Evaluation II

Impact of Number of Machines in Performances of [**mrmodelSODA10**]

	machines	map time (sec)	reduce time (sec)
cnr-2000	1	49	29
cnr-2000	2	44	29
cnr-2000	3	59	29
in-2004	1	210	47
in-2004	2	194	47
in-2004	3	209	52

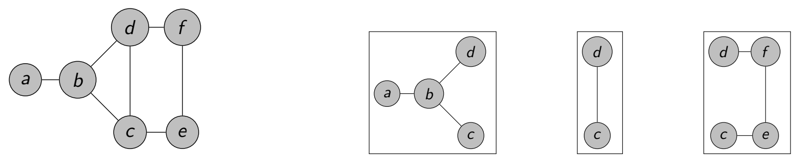
Implications of changes in the number of machines, with $k = 3$: increasing the number of machines *might* increase overall computation time (w.r.t. running more map or reduce instances on the same machine)

Preliminary Experimental Evaluation III

Number of Rounds in [**filteringSPAA11**]

Let us assume, in the r-th round:

- $|E| > k$;
- each of the subgraphs G_i is a tree or a forest.



G G_1 G_2 G_3

input graph $=$ output graph, and the r-th is a "void" round.

Preliminary Experimental Evaluation IV

Number of Rounds in [**filteringSPAA11**]

(Graph instances having same c value 0.18)

	c'	expected rounds	average rounds[1]
cnr-2000	0.03	8	8.00
cnr-2000	0.05	5	7.33
cnr-2000	0.15	2	3.00
in-2004	0.03	6	6.00
in-2004	0.05	4	4.00
in-2004	0.15	2	2.00

We noticed some few "void" round occurrences.
(Partitioning using a random hash function)

Simulation of PRAMs via MapReduce I

mrmodelSODA10; **MUD10**; **G10**

(1) CRCW PRAM. via *memory-bound MapReduce* framework.

(2) CREW PRAM. via $\mathcal{DMRC}$:
(PRAM) $O(S^{2-2\epsilon})$ total memory, $O(S^{2-2\epsilon})$ processors and T time.
(MapReduce) $O(T)$ rounds, $O(S^{2-2\epsilon})$ reducer instances.

(3) EREW PRAM. via MUD model of computation.

PRAM Algorithms for the MST

- CRCW PRAM algorithm [**MST96**]
  (randomized)
  $O(\log n)$ time, $O(N)$ work $\rightarrow$ work-optimal
- CREW PRAM algorithm [**JaJa92**]
  $O(\log^2 n)$ time, $O(n^2)$ work $\rightarrow$ work-optimal if $N = O(n^2)$.
- EREW PRAM algorithm [**Johnson92**]
  $O(\log^{\frac{3}{2}} n)$ time, $O(N \log^{\frac{3}{2}} n)$ work.
- EREW PRAM algorithm [**wtMST02**]
  (randomized)
  $O(N)$ total memory, $O(\frac{N}{\log n})$ processors.
  $O(\log n)$ time, $O(N)$ work $\rightarrow$ work-time optimal.

Simulation of CRCW PRAM with CREW PRAM: $\Omega(\log S)$ steps.

Simulation of [**wtMST02**] via MapReduce I

The algorithm is *external* (for dense and sparse graphs).

Simulate the algorithm in [**wtMST02**] using CREW$\rightarrow$MapReduce.

- the memory requirement of any map or reduce computation is

$$O(\log n) = O(n^{1-\epsilon}) \tag{4}$$

  for some

$$1 - \log\log n \geq \epsilon > 0 \tag{5}$$

- the algorithm works in $O(\log n)$ rounds.

Summary

	[**mrmodelSODA10**]	[**filteringSPAA11**]	Simulation
	G is c-dense, and $c \geq c' > 0$		
	if $k = n^{\frac{c'}{2}}$, whp	if $k = n^{1+c'}$	
Memory	$O(N^{1-\epsilon})$	$O(n^{1+c'}) = O(N^{1-\epsilon})$	$O(\log n) = O(n^{1-\epsilon})$
Rounds	2	$\lceil \frac{c}{c'} \rceil = O(1)$	$O(\log n)$

Table: Space and Time complexity of algorithms discussed so far.

Borůvka MST algorithm I

boruvka26

Classical model of computation algorithm

procedure `Borůvka MST`$(G(V,E))$:
$T \to V$
while $|T| < n-1$ **do**
  for all connected component C in T **do**
    $e \to$ the smallest-weight edge from C to another component in T
    if $e \notin T$ **then**
      $T \to T \cup \{e\}$
    end if
  end for
end while

Borůvka MST algorithm II

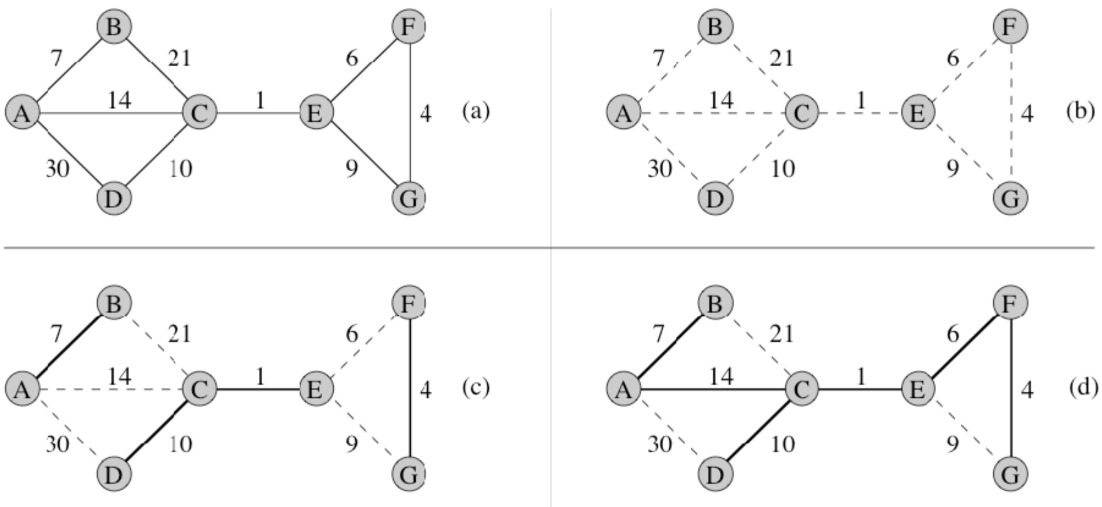


Figure: An example of Borůvka algorithm execution.

Random Mate CC algorithm I

rm91

CRCW PRAM model of computation algorithm

procedure `Random Mate CC`$(G(V, E))$:
for all $v \in V$ **do** $cc(v) \rightarrow v$ **end for**
while there are edges connecting two CC in G (*live*) **do**
  for all $v \in V$ **do** gender[v] $\rightarrow$ rand({M, F}) **end for**
  for all live $(u, v) \in V$ **do**
    $cc(u)$ is M $\wedge$ $cc(v)$ is F ? $cc(cc(u)) \rightarrow cc(v)$: $cc(cc(v)) \rightarrow cc(u)$
  end for
  for all $v \in E$ **do** $cc(v) \rightarrow cc(cc(v))$ **end for**
end while

Random Mate CC algorithm II

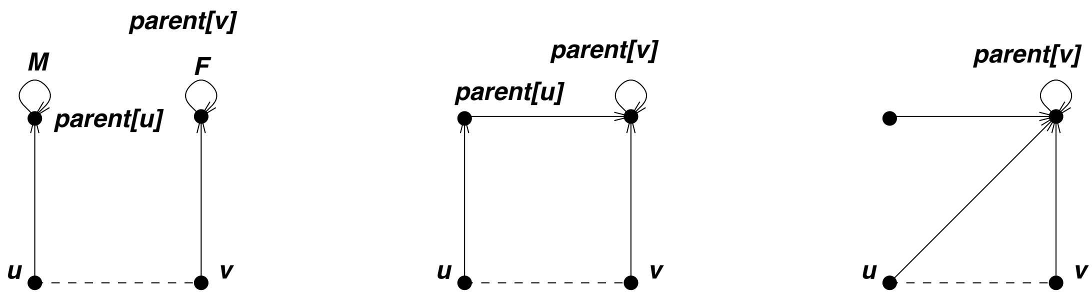
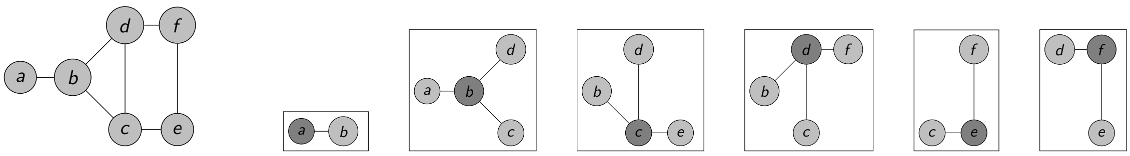


Figure: An example of Random Mate algorithm step.

Borůvka + Random Mate I

Let us consider again the labeling function $cc : V \to V$

(1) Map Step i (Borůvka).
Given an edge $(u, v) \in E$, the result of the mapping consists in two *key* : *value* pairs $cc(u) : (u, v)$ and $cc(v) : (u, v)$.



G $\quad G_1 \quad G_2 \quad G_3 \quad G_4 \quad G_5 \quad G_6$

Borůvka + Random Mate II

(2) Reduce Step i (Borůvka).
For each subgraph G_i, execute one iteration of the Borůvka algorithm.

Let T be the output of i-th Borůvka iteration.
Execute r_i Random Mate rounds, feeding the first one with T.

(3) Round $i + j$ (Random Mate).
Use a MapReduce implementation [pb10] of Random Mate algorithm and update the function cc.

- if there are no more live edges, the algorithm ends (T is the MST of the input graph G)
- otherwise $\to$ start a new Borůvka round

Borůvka + Random Mate III

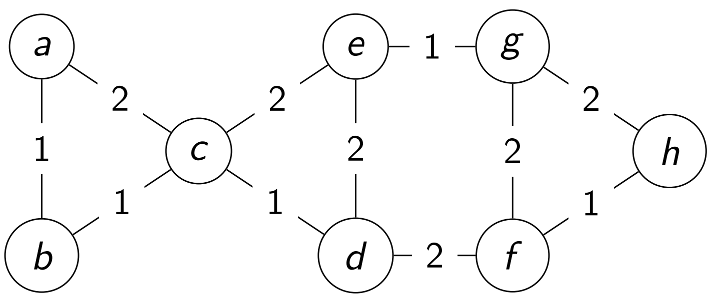
Two extremal cases:

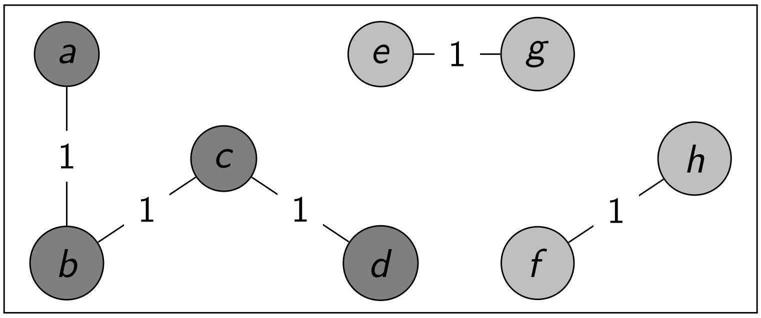
- output of first Borůvka round is connected
  $\rightarrow O(\log n)$ Random Mate rounds, and algorithm ends.
- output of each Borůvka round is a matching
  $\rightarrow \forall i, r_i = 1$ Random Mate round
  $\rightarrow O(\log n)$ Borůvka rounds, and algorithm ends.

Therefore

- it works in $O(\log^2 n)$ rounds;
- example working in $\approx \frac{1}{4}\log^2 n$

Borůvka + Random Mate IV





Conclusions

Work in progress for an *external* implementation of the algorithm (for dense and sparse graphs).

- the worst case seems to rely on a certain kind of structure in the graph, difficult to appear in realistic graphs
- need of more experimental work to confirm it

Is there any external constant round algorithm for connected components and MST in sparse graphs?

Maybe under certain (and hopefully realistic) assumptions.

Overview...

- MapReduce was developed by Google, and later implemented in Apache Hadoop
- Hadoop is easy to install and use, and Amazon sells computational power at really low prices
- Theoretical models have been presented, but so far there is no established theoretical framework for analysing MapReduce algorithms
- Several “similar” systems (Dryad, S4, Pregel) have been presented, but are not diffused as MapReduce/Hadoop... also because...

The End... I told you from the beginning...

> *"The beauty of MapReduce is that any programmer can understand it, and its power comes from being able to harness thousands of computers behind that simple interface"*

David Patterson