INFO-H515: BIG DATA: DISTRIBUTED MANAGEMENT

Lecture 6: Parallel Processing

Dimitris Sacharidis

2023–2024

Our story so far:

It is possible to analyze huge data sets by exploiting data parallellism:

- partition and distribute the data over a cluster consisting of many machines
- machines operate in parallel on part of the data and communicate over a network to compute the final result

Our story so far:

It is possible to analyze huge data sets by exploiting data parallellism:

- partition and distribute the data over a cluster consisting of many machines
- machines operate in parallel on part of the data and communicate over a network to compute the final result

- What are performance bottlenecks in this context?
- Are there limitations to data parallellism?

LECTURE OUTLINE

Where's the bottleneck?

The Bulk Synchronous Parallel (BSP) Model

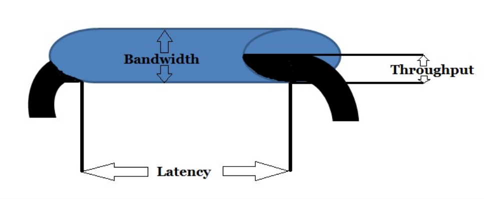
BSP Application: Think Like a Vertex

Speedup and Scaleup

Scalability, but at what COST?

WHERE'S THE BOTTLENECK?

DEFINITION



Definition

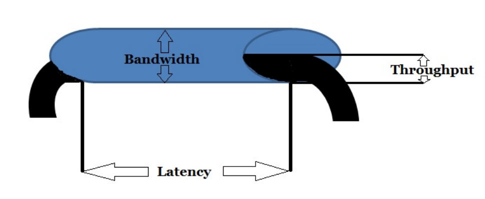
Throughput is the total amount of work done in a given time

Examples:

- 1 TB of data analyzed in 1h $\approx$ 291 MB / s throughput
- 100 MB copied from disk A to disk B in 10s = 10 Mb / s throughput

Figure source: `http://perfmatrix.blogspot.be/2016/12/latency-bandwidth-throughput-responsetime.html`

DEFINITION



Definition

Latency, also known as response time is the time between the start and completion of an event

Examples:

- If it takes 1h to analyze 1 terabyte completely, the latency is 1h.
- A disk seek on a rotational disk takes 10ms, which equals the latency.

Figure source:`http://perfmatrix.blogspot.be/2016/12/latency-bandwidth-throughput-responsetime.html`

IMPORTANT LATENCY NUMBERS

L1 cache reference	0.5 ns	
Branch mispredict	5 ns	
L2 cache reference	7 ns	
Mutex lock/unlock	25 ns	
Main memory reference	100 ns	
Compress 1K bytes with Zippy	3,000 ns	= 3 μs
Send 2K bytes over 1 Gbps network	10,000 ns	= 10 μs
Read 4K randomly from SSD*	150,000 ns	= 150 μs
Read 1 MB sequentially from memory	250,000 ns	= 250 μs
Round trip within same datacenter	500,000 ns	= 500 μs
Read 1 MB sequentially from SSD*	1,000,000 ns	= 1 **ms**
Disk seek	10,000,000 ns	= 10 **ms**
Read 1 MB sequentially from disk	20,000,000 ns	= 20 **ms**
Send packet US $\rightarrow$ Europe $\rightarrow$ US	150,000,000 ns	= 150 **ms**

IMPORTANT LATENCY NUMBERS

L1 cache reference	0.5 ns	
Branch mispredict	5 ns	
L2 cache reference	7 ns	
Mutex lock/unlock	25 ns	
Main memory reference	100 ns	
Compress 1K bytes with Zippy	3,000 ns	= 3 μs
Send 2K bytes over 1 Gbps network	10,000 ns	= 10 μs
Read 4K randomly from SSD*	150,000 ns	= 150 μs
Read 1 MB sequentially from memory	250,000 ns	= 250 μs
Round trip within same datacenter	500,000 ns	= 500 μs
Read 1 MB sequentially from SSD*	1,000,000 ns	= 1 ms
Disk seek	10,000,000 ns	= 10 ms
Read 1 MB sequentially from disk	20,000,000 ns	= 20 ms
Send packet US $\rightarrow$ Europe $\rightarrow$ US	150,000,000 ns	= 150 ms

IMPORTANT LATENCY NUMBERS

L1 cache reference	0.5 ns	
Branch mispredict	5 ns	
L2 cache reference	7 ns	
Mutex lock/unlock	25 ns	
Main memory reference	100 ns	
Compress 1K bytes with Zippy	3,000 ns	= 3 μs
Send 2K bytes over 1 Gbps network	10,000 ns	= 10 μs
Read 4K randomly from SSD*	150,000 ns	= 150 μs
Read 1 MB sequentially from memory	250,000 ns	= 250 μs
Round trip within same datacenter	500,000 ns	= 500 μs
Read 1 MB sequentially from SSD*	1,000,000 ns	= 1 ms
Disk seek	10,000,000 ns	= 10 ms
Read 1 MB sequentially from disk	20,000,000 ns	= 20 ms
Send packet US $\rightarrow$ Europe $\rightarrow$ US	150,000,000 ns	= 150 ms

WHAT DOES THIS MEAN?

To get a better intuition of the orders of magnitude differences in these latencies, let's humanize the duration.

Method:

- multiply all durations by a billion
- then, we can associate each latency number to a human activity

HUMANIZED LATENCY NUMBERS

Humanized durations grouped by magnitude:

Minute

L1 cache reference	0.5 s	One heartbeat
Branch mispredict	5 s	Yawn
L2 cache reference	7 s	Long yawn
Mutex lock/unlock	25 s	Making a coffee

Hour

Main memory reference	100 s	Brushing your teeth
Compress 1K bytes with Zippy	50 min	One TV show episode

HUMANIZED LATENCY NUMBERS

Day

Send 2K bytes over 1 Gbps network	5.5 hr	Workday afternoon

Week

Read 4K randomly from SSD*	1.7 days	A weekend
Read 1 MB sequentially from memory	2.9 days	A long weekend
Round trip within same datacenter	5.8 days	A small vacation
Read 1 MB sequentially from SSD*	11.6 days	Two weeks

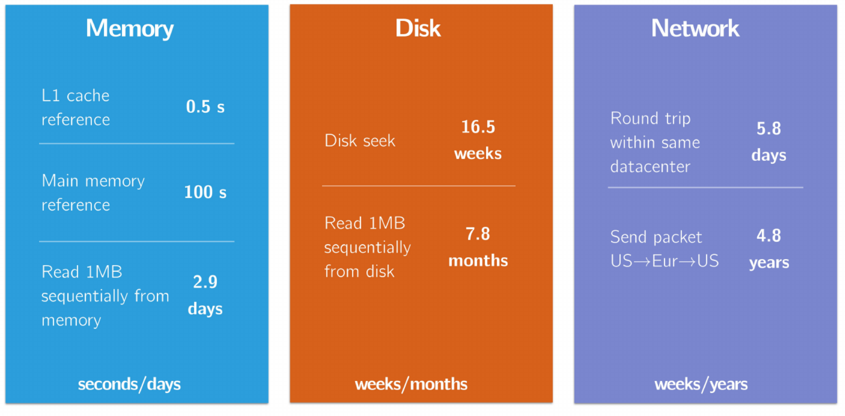
HUMANIZED LATENCY NUMBERS

Year

Disk seek	16.5 weeks	A semester at ULB
Read 1 MB sequentially from disk	7.8 months	Almost a full pregnancy

Decade

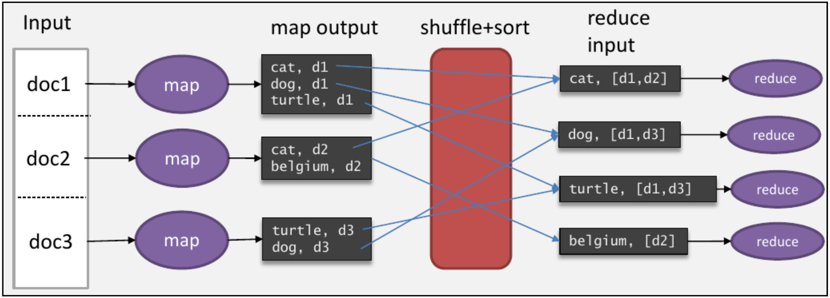
Send packet US $\rightarrow$ Europe $\rightarrow$ US	4.8 years	The length of your studies



Fast Slow Slowest

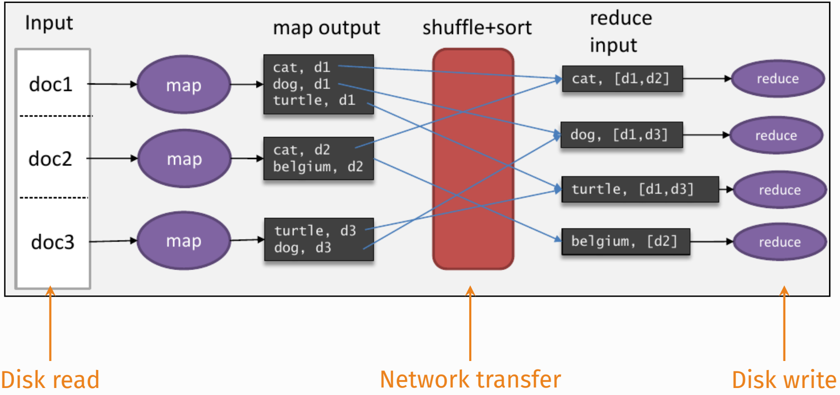
LATENCY IN MAP/REDUCE

Spot the important latencies in a single M/R job:



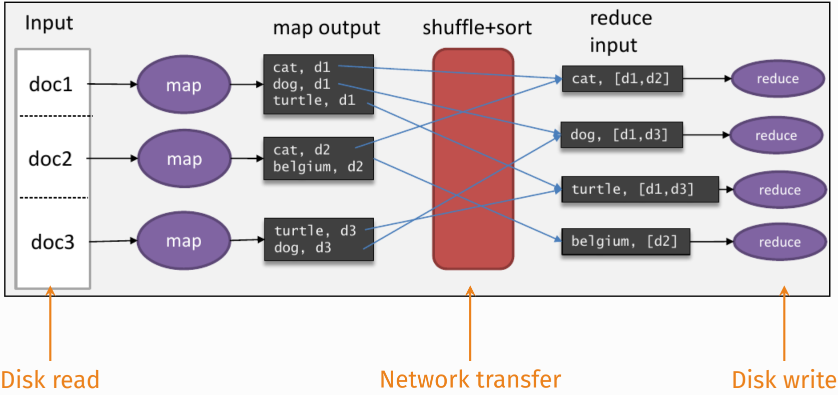
LATENCY IN MAP/REDUCE

Spot the important latencies in a single M/R job:



LATENCY IN MAP/REDUCE

Spot the important latencies in a single M/R job:



Note: latencies accumulate if you have a sequence of jobs!

LATENCY IN SPARK

Spot the important latencies in a Spark program:

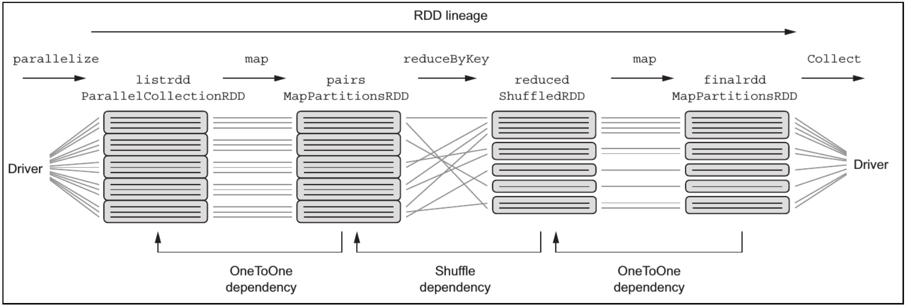


Figure source: Spark in Action book

LATENCY IN SPARK

Spot the important latencies in a Spark program:

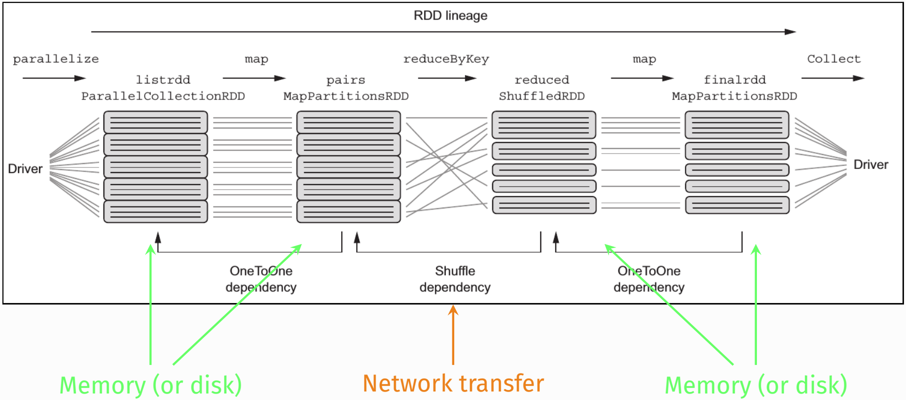
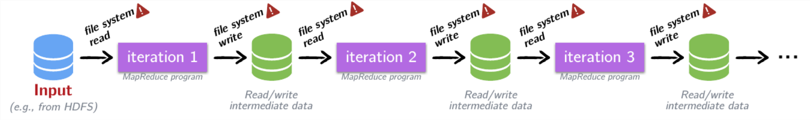


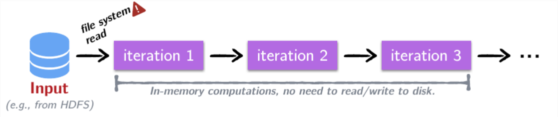
Figure source: Spark in Action book

SOME CONCLUSIONS:

M/R



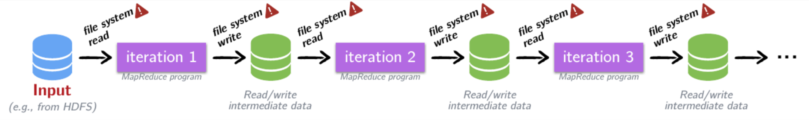
Spark



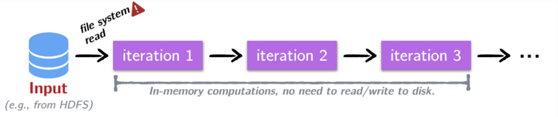
- Spark is better for iterative algorithms (typical in Machine Learning).
- Try and avoid operations that cause a shuffle in both M/R and Spark. For instance, prefer map-only M/R jobs; be careful how you partition in Spark.

SOME CONCLUSIONS:

M/R

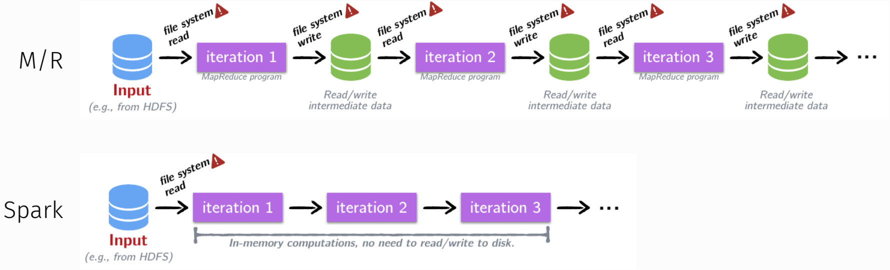


Spark



- Spark is better for iterative algorithms (typical in Machine Learning).
- Try and avoid operations that cause a shuffle in both M/R and Spark. For instance, prefer map-only M/R jobs; be careful how you partition in Spark.
  Question: Is this always possible?

SOME CONCLUSIONS:



- Spark is better for iterative algorithms (typical in Machine Learning).
- Try and avoid operations that cause a shuffle in both M/R and Spark. For instance, prefer map-only M/R jobs; be careful how you partition in Spark.

  Question: Is this always possible? Answer: let's investigate!

THE BULK SYNCHRONOUS PARALLEL (BSP) MODEL

WHAT IS A PARALLEL COMPUTER ?

Definition

A parallel computer consists of a set of processors (such as a cluster of PCs) that work together to solve a computational problem.

Two types:

- Shared memory parallel computers (e.g., multi-core computer, a super-computer)
- Shared-nothing cluster of machines, a.k.a. distributed-memory parallel computer (e.g., a Big Data compute cluster).

WHAT IS A PARALLEL COMPUTER ?

Definition

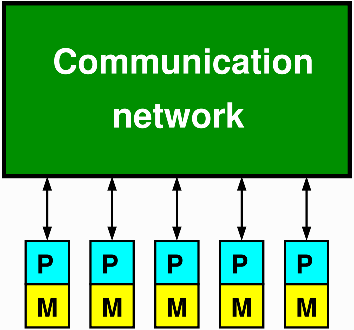
A parallel computer consists of a set of processors (such as a cluster of PCs) that work together to solve a computational problem.

Two types:

- Shared memory parallel computers (e.g., multi-core computer, a super-computer)
- Shared-nothing cluster of machines, a.k.a. distributed-memory parallel computer (e.g., a Big Data compute cluster).

Question: How do you investigate the computational complexity of a problem that is meant to be solved by a parallel computer?

PARALLEL COMPUTER: ABSTRACT MODEL



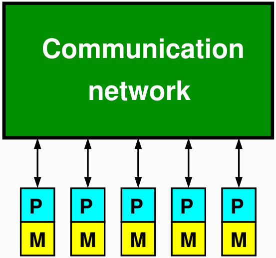
Bulk Synchronous Parallel (BSP) computer

Proposed by Leslie Valiant, 1989

Purpose:

- provide a simple yet practical framework for general-purpose parallel computing;
- in order to support the creation of architecture-independent and scalable parallel software.

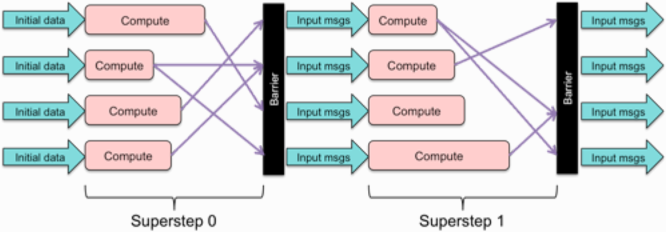
BSP COMPUTER: MODEL OF PARALLEL COMPUTER



- A BSP computer consists of a collection of processors, each with its own memory. It is hence a distributed memory computer.
- Point to point communication between processors is enabled by a communication network, which is treated as a black box.
- Access to own memory is fast, to remote memory slower.
- Uniform-time access to all remote memories.

COMPUTATION IN THE BSP COMPUTER

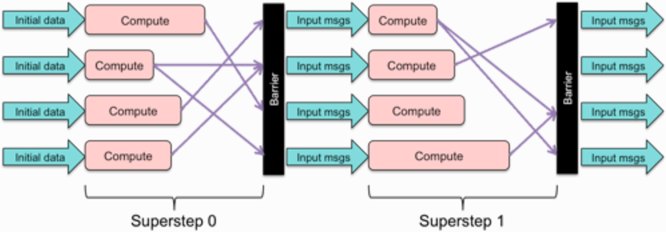
The execution of an algorithm on a BSP consists of a sequence of supersteps.



Each superstep consists of:

COMPUTATION IN THE BSP COMPUTER

The execution of an algorithm on a BSP consists of a sequence of supersteps.

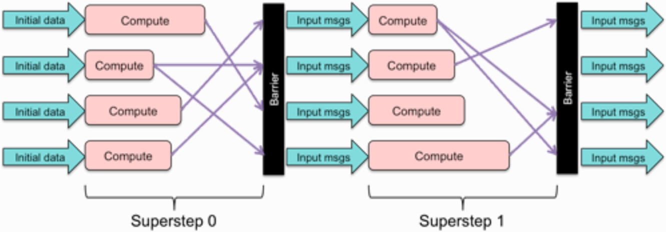


Each superstep consists of:

- A computation phase: processors compute asynchronously on data in local memory

COMPUTATION IN THE BSP COMPUTER

The execution of an algorithm on a BSP consists of a sequence of supersteps.

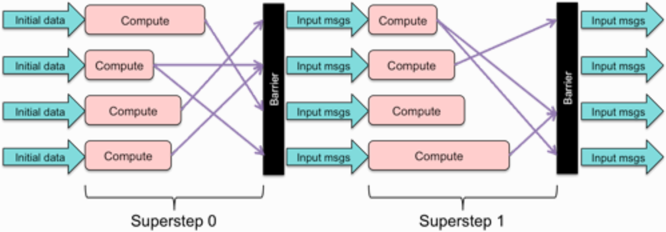


Each superstep consists of:

- A computation phase: processors compute asynchronously on data in local memory
- A communication phase: during which processors can send data to other processors

COMPUTATION IN THE BSP COMPUTER

The execution of an algorithm on a BSP consists of a sequence of supersteps.

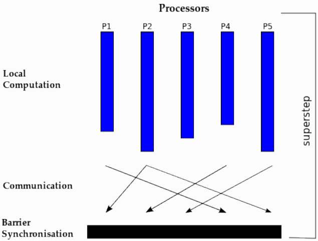


Each superstep consists of:

- A computation phase: processors compute asynchronously on data in local memory
- A communication phase: during which processors can send data to other processors
- A synchronization barrier: processors synchronize, and wait until all processors have finished computation & communication.

  Once the barrier is passed, all processors have received all data sent to them in the communication phase, and this data is now hence locally available for the next superstep.

BSP COST MODEL

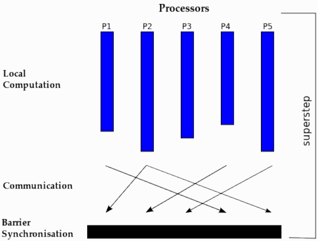


- Basic arithmetic operations and local memory accesses have unit cost (e.g., 1 time unit).
- Cost C_i of a superstep i:

$$C_i := w_i + h_i \cdot g + l$$

[1]Models set-up cost of communications and synchronization

BSP COST MODEL



- Basic arithmetic operations and local memory accesses have unit cost (e.g., 1 time unit).
- Cost C_i of a superstep i:

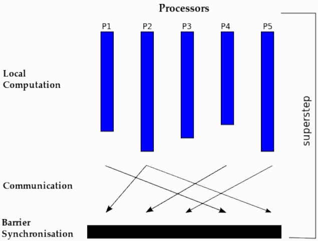
$$C_i := w_i + h_i \cdot g + l$$

where:

  - w_i is the maximum number of local operations performed by a processor in superstep i;

[1]Models set-up cost of communications and synchronization

BSP COST MODEL



- Basic arithmetic operations and local memory accesses have unit cost (e.g., 1 time unit).
- Cost C_i of a superstep i:

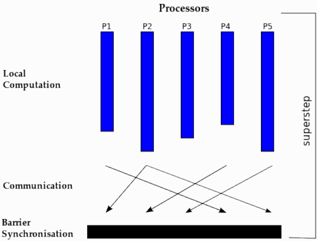
$$C_i := w_i + h_i \cdot g + l$$

where:

  - w_i is the maximum number of local operations performed by a processor in superstep i;
  - h_i is the maximum number of data units sent or received by a processor during superstep i;

[1]Models set-up cost of communications and synchronization

BSP COST MODEL



- Basic arithmetic operations and local memory accesses have unit cost (e.g., 1 time unit).
- Cost C_i of a superstep i:

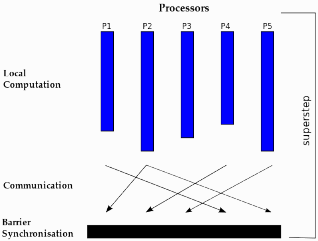
$$C_i := \boxed{w_i} + \boxed{h_i} \cdot \boxed{g} + \boxed{l}$$

where:

  - w_i is the maximum number of local operations performed by a processor in superstep i;
  - h_i is the maximum number of data units sent or received by a processor during superstep i;
  - g is the throughput ratio (time required to communicate one data unit); and

[1] Models set-up cost of communications and synchronization

BSP COST MODEL



- Basic arithmetic operations and local memory accesses have unit cost (e.g., 1 time unit).
- Cost C_i of a superstep i:

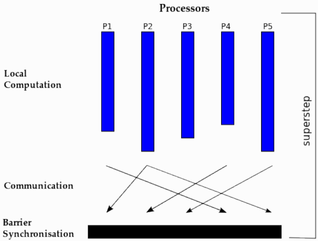
$$C_i := \boxed{w_i} + \boxed{h_i} \cdot \boxed{g} + \boxed{l}$$

where:

  - w_i is the maximum number of local operations performed by a processor in superstep i;
  - h_i is the maximum number of data units sent or received by a processor during superstep i;
  - g is the throughput ratio (time required to communicate one data unit); and
  - l is the communication latency[1].

[1] Models set-up cost of communications and synchronization

BSP COST MODEL



Question: What does this cost represent?

- Basic arithmetic operations and local memory accesses have unit cost (e.g., 1 time unit).
- Cost C_i of a superstep i:

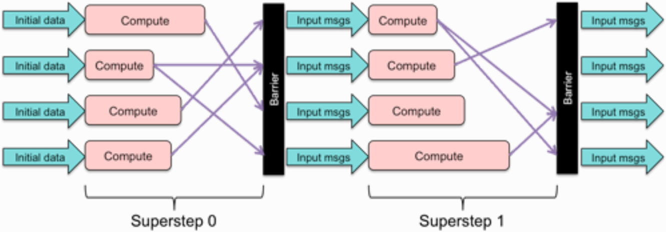
$$C_i := w_i + h_i \cdot g + l$$

where:

  - w_i is the maximum number of local operations performed by a processor in superstep i;
  - h_i is the maximum number of data units sent or received by a processor during superstep i;
  - g is the throughput ratio (time required to communicate one data unit); and
  - l is the communication latency[1].

[1] Models set-up cost of communications and synchronization

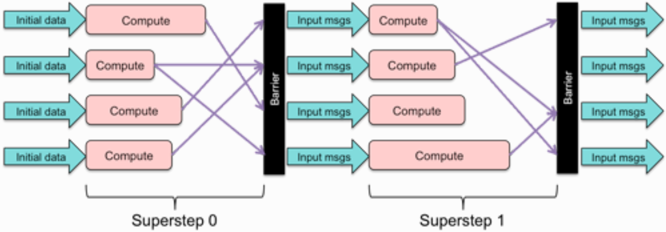
BSP COST MODEL



Cost C of a computation consisting of S supersteps:

$$
\begin{aligned}
C &= \sum_{i=1}^{S}(w_i + h_i \cdot g + l) \\
&= (\sum_{i=1}^{S} w_i) + (\sum_{i=1}^{S} h_i) \cdot g + S \cdot l \\
&= W + H \cdot g + S \cdot l
\end{aligned}
$$

BSP ALGORITHM DESIGN



Cost C of a computation consisting of S supersteps:

$$C = W + H \cdot g + S \cdot l$$

Upper/lower bounds are known for the BSP cost of many problems.

- If you every need a smart algorithm for solving a problem in distributed fashion, have a look at the literature!
- An then try and implement the BSP algorithm in your favorite big data framework.

BSP ALGORITHM DESIGN

When designing a BSP algorithm, there is often a trade-off between communication and synchronization.

Simplistic example: communicate a single value to all processors.

- Method 1: broadcast the value to all processors.

$$H = O(p) \quad S = O(1)$$

- Method 2: organize the p processors in a balanced binary tree.

$$H = S = O(\log p)$$

BSP ALGORITHM DESIGN

When designing a BSP algorithm, there is often a trade-off between communication and synchronization.

Simplistic example: communicate a single value to all processors.

- Method 1: broadcast the value to all processors.

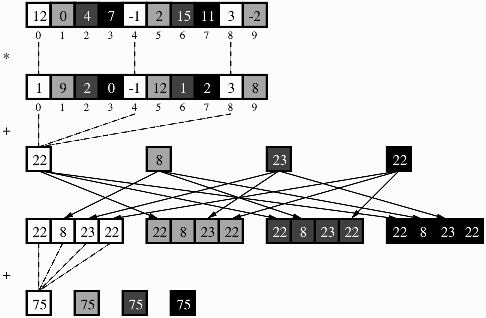
$$H = O(p) \quad S = O(1)$$

- Method 2: organize the p processors in a balanced binary tree.

$$H = S = O(\log p)$$

Question: **Which of these two is "the most sequential"?**

BSP ALGORITHM DESIGN



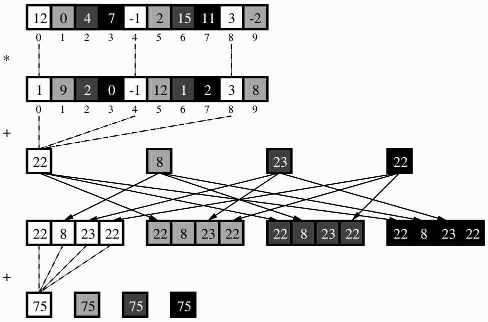
Parallel inner product over 4 processors.

Other example

Inner-product of two vectors $\vec{x}$ and $\vec{y}$

- Distribute the elements of $\vec{x}$ and $\vec{y}$ over the p processors such that $\vec{x}_i$ and $\vec{y}_i$ are on the same processors, for each i.
- Locally multiply and sum the data at each processor
- Broadcast these values
- Each processor then computes the final result.

BSP ALGORITHM DESIGN



Parallel inner product over 4 processors.

Other example

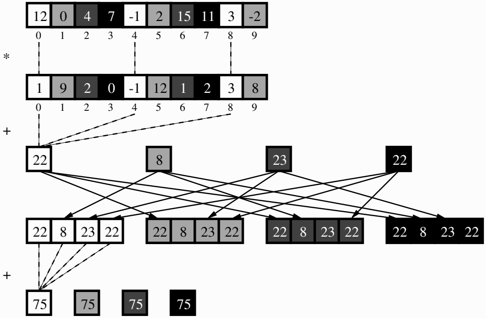
Inner-product of two vectors $\vec{x}$ and $\vec{y}$

- Distribute the elements of $\vec{x}$ and $\vec{y}$ over the p processors such that $\vec{x}_i$ and $\vec{y}_i$ are on the same processors, for each i.
- Locally multiply and sum the data at each processor
- Broadcast these values
- Each processor then computes the final result.

Cost analysis:

- Superstep 1:
  - Local work to multiply and sum: $w_1 = \left\lceil \frac{n}{p} \right\rceil$
  - Broadcast: $h_1 = p - 1$

BSP ALGORITHM DESIGN



Parallel inner product over 4 processors.

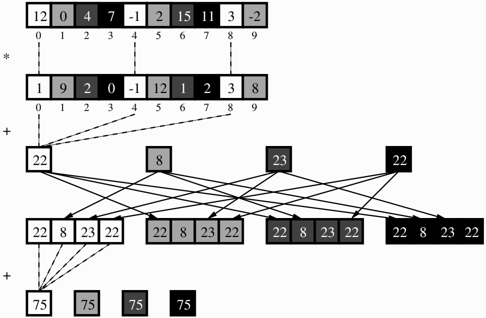
Other example

Inner-product of two vectors $\vec{x}$ and $\vec{y}$

- Distribute the elements of $\vec{x}$ and $\vec{y}$ over the p processors such that $\vec{x}_i$ and $\vec{y}_i$ are on the same processors, for each i.
- Locally multiply and sum the data at each processor
- Broadcast these values
- Each processor then computes the final result.

Cost analysis:

- Superstep 2:
  - Local work to sum partial results: $w_2 = p - 1$
  - No communication: $h_2 = 0$



Parallel inner product over 4 processors.

Other example

Inner-product of two vectors $\vec{x}$ and $\vec{y}$

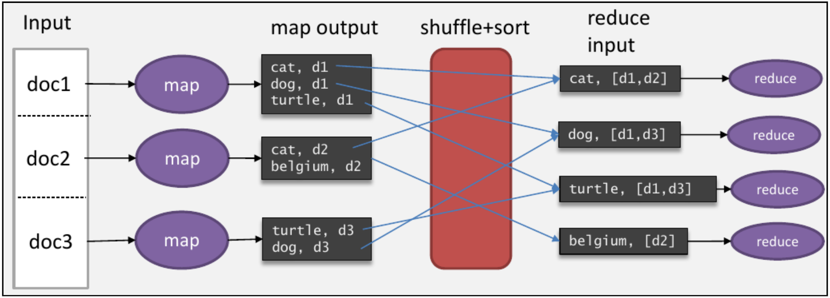
- Distribute the elements of $\vec{x}$ and $\vec{y}$ over the p processors such that $\vec{x}_i$ and $\vec{y}_i$ are on the same processors, for each i.
- Locally multiply and sum the data at each processor
- Broadcast these values
- Each processor then computes the final result.

Cost analysis:

- Total cost = $\underbrace{\left(\left\lceil \frac{n}{p} \right\rceil + p - 1\right)}_{W} + \underbrace{(p-1)}_{H} g + \underbrace{2}_{S} l$

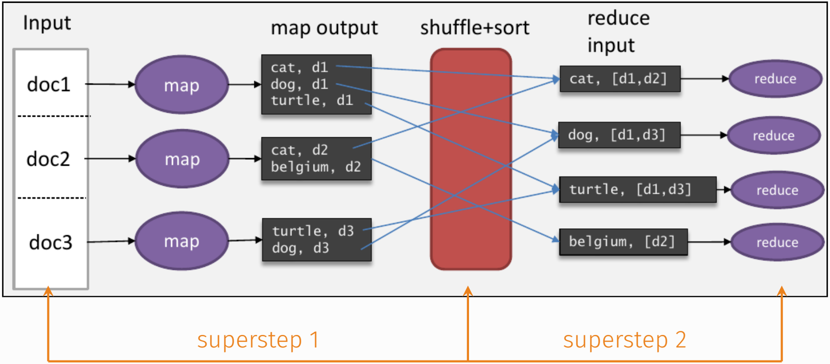
BSP AND MAP/REDUCE

Spot the supersteps in a single M/R job.:



BSP AND MAP/REDUCE

Spot the supersteps in a single M/R job.:



- One M/R job consists of two supersteps: 1 for the map phase; 1 for the reduce phase. Sequences of M/R jobs hence give you a way to implement a BSP algorithm.

BSP AND SPARK

Spot the supersteps in a Spark program:

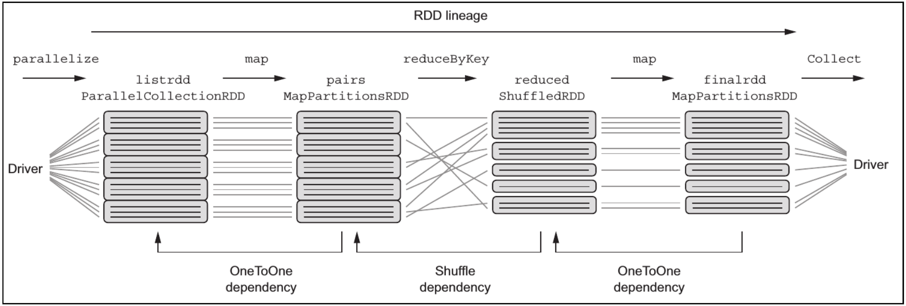


Figure source: Spark in Action book

BSP AND SPARK

Spot the supersteps in a Spark program:

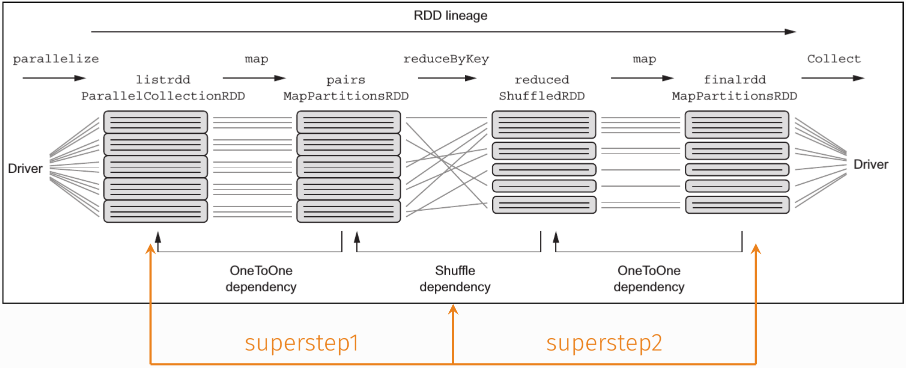
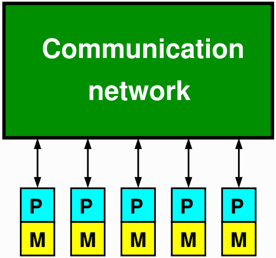


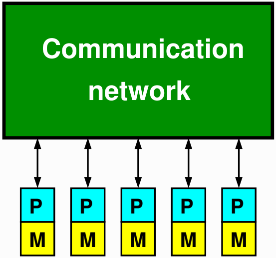
Figure source: Spark in Action book

BSP: CONCLUSION



- A BSP computer models a distributed memory parallel computer.
- It allows analysis of parallel algorithms, taking into account the cost of (parallel) local computation; communication; and synchronization.
- The BSP computer model is very general. Therefore algorithms designed for a BSP computer are portable: they can be run efficiently on many different parallel computers/programming frameworks.

BSP: CONCLUSION



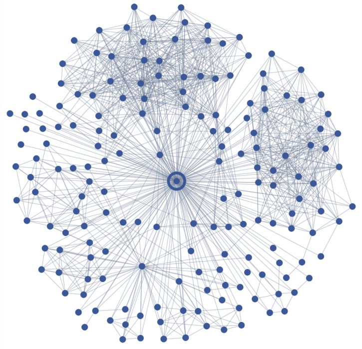
- A BSP computer models a distributed memory parallel computer.
- It allows analysis of parallel algorithms, taking into account the cost of (parallel) local computation; communication; and synchronization.
- The BSP computer model is very general. Therefore algorithms designed for a BSP computer are portable: they can be run efficiently on many different parallel computers/programming frameworks.

An important comment:

- The “Communication network” of a BSP computer need not be a computer network; it is just a communication channel.
- If you interpret the “communication network” as a shared memory (with remote memory access as fast as local memory), then this models a shared-memory parallel computer (PRAM).

BSP APPLICATION: THINK LIKE A VERTEX

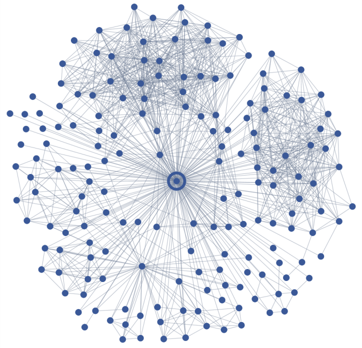
BIG GRAPH ANALYTICS



Lots of Big Data analytics involve analysis of (big) graphs:

- Social networks
- Biological networks
- Mobile call networks
- World Wide Web
- Customer-merchant graphs (Amazon, Ebay)
- …

BIG GRAPH ANALYTICS



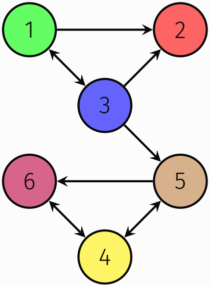
Lots of Big Data analytics involve analysis of (big) graphs:

- Social networks
- Biological networks
- Mobile call networks
- World Wide Web
- Customer-merchant graphs (Amazon, Ebay)
- ...

Applications:

- Recommendation
- PageRank
- Web Search
- Cyber Security
- Fraud detection
- Clustering

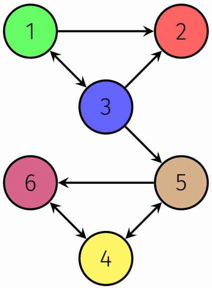
EXAMPLE: PAGERANK



Example:

PageRank, Google's famous algorithm for measuring the authority of a webpage based on the underlying network of hyperlinks.

EXAMPLE: PAGERANK



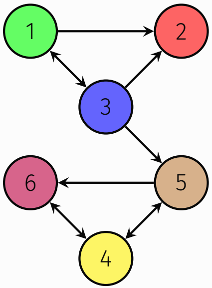
Example:
PageRank, Google's famous algorithm for measuring the authority of a webpage based on the underlying network of hyperlinks.

The PageRank p_i of a page i is given by

$$p_i = \sum_{j \in B_i} \frac{p_j}{N_j}$$

where B_i is the set of pages linking to i and N_j is the number of links on page j.

EXAMPLE: PAGERANK



Example:
PageRank, Google's famous algorithm for measuring the authority of a webpage based on the underlying network of hyperlinks.

The PageRank p_i of a page i is given by

$$p_i = \sum_{j \in B_i} \frac{p_j}{N_j}$$

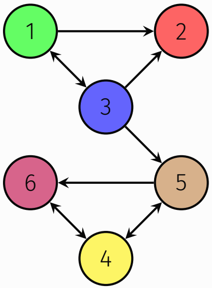
where B_i is the set of pages linking to i and N_j is the number of links on page j.

$$p_1 = \frac{1}{3}p_3 \qquad p_2 = \frac{1}{2}p_1 + \frac{1}{3}p_3$$

$$p_3 = \frac{1}{2}p_1 \qquad p_4 = \frac{1}{2}p_5 + \frac{1}{2}p_6$$

$$p_5 = \frac{1}{3}p_3 + \frac{1}{2}p_4 \qquad p_6 = \frac{1}{2}p_4 + \frac{1}{2}p_5$$

EXAMPLE: PAGERANK



Example:
PageRank, Google's famous algorithm for measuring the authority of a webpage based on the underlying network of hyperlinks.

The PageRank p_i of a page i is given by

$$p_i = \sum_{j \in B_i} \frac{p_j}{N_j}$$

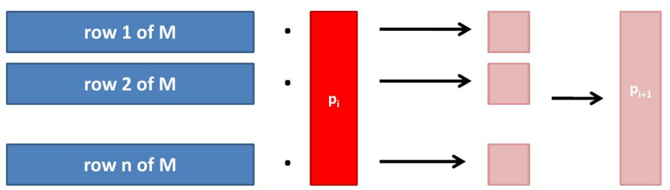
where B_i is the set of pages linking to i and N_j is the number of links on page j.

$$\begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{bmatrix} = \begin{bmatrix} 0 & 0 & \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & \frac{1}{3} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 \end{bmatrix} \cdot \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{bmatrix}$$

EXAMPLE: PAGERANK

Textbook approach to PageRank in M/R or Spark:

- From the web hyperlink graph one can construct a matrix M that essentially captures the transition probabilities $M_{i,j} = \frac{1}{N_j}$ from node j to node i.[2]
- The pagerank can then be obtained by multiplying an initial PageRank vector by M (power iteration): $\vec{p} = M^k \cdot \vec{p_0}$

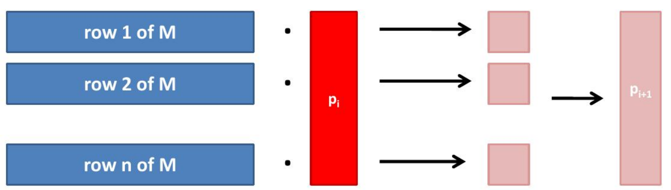


[2]For technical reasons, the actual matrix used is slightly different.

EXAMPLE: PAGERANK

Textbook approach to PageRank in M/R or Spark:

- From the web hyperlink graph one can construct a matrix M that essentially captures the transition probabilities $M_{i,j} = \frac{1}{N_j}$ from node j to node i.[2]
- The pagerank can then be obtained by multiplying an initial PageRank vector by M (power iteration): $\vec{p} = M^k \cdot \vec{p_0}$
- One can argue that the algorithm is not immediately clear. Also, where are the communication/synchronization bottlenecks?



[2] For technical reasons, the actual matrix used is slightly different.

VERTEX-CENTRIC BSP

BSP Version of Pagerank:

Consider that each vertex is a (virtual) processor which (locally) knows its outgoing edges. Then, for k supersteps each vertex operates as follows:

VERTEX-CENTRIC BSP

BSP Version of Pagerank:

Consider that each vertex is a (virtual) processor which (locally) knows its outgoing edges. Then, for k supersteps each vertex operates as follows:

- if $k = 0$, then initialize pagerank to a random number else update pagerank based on neighbour's messages

```
pagerank = sum(received-messages);
```

BSP Version of Pagerank:

Consider that each vertex is a (virtual) processor which (locally) knows its outgoing edges. Then, for k supersteps each vertex operates as follows:

- if $k = 0$, then initialize pagerank to a random number else update pagerank based on neighbour's messages

```
pagerank = sum(received-messages);
```

- send updated pagerank to each neighbour

```
send (pagerank/number of links) to all neighbours
```

BSP Version of Pagerank:

Consider that each vertex is a (virtual) processor which (locally) knows its outgoing edges. Then, for k supersteps each vertex operates as follows:

- if $k = 0$, then initialize pagerank to a random number else update pagerank based on neighbour's messages

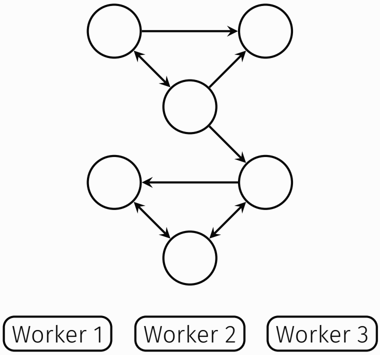
```
pagerank = sum(received-messages);
```

- send updated pagerank to each neighbour

```
send (pagerank/number of links) to all neighbours
```

Simple! ...But we don't have as many processors as vertices.

VERTEX-CENTRIC BSP

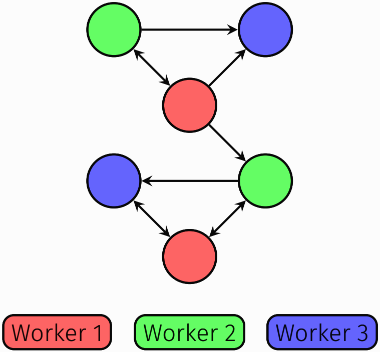


Problem: We have more vertices than real processors (workers)

Solution:

- Assign each vertex to a real processor (worker). E.g. by hash-partitioning, or some more clever form of partitioning.
- In each superstep, the real workers *accumulate* the messages sent by the vertices to other vertices, and communicate these to the corresponding workers.

VERTEX-CENTRIC BSP

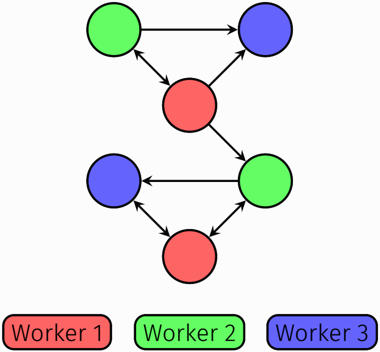


Problem: We have more vertices than real processors (workers)

Solution:

- Assign each vertex to a real processor (worker). E.g. by hash-partitioning, or some more clever form of partitioning.
- In each superstep, the real workers *accumulate* the messages sent by the vertices to other vertices, and communicate these to the corresponding workers.

VERTEX-CENTRIC BSP



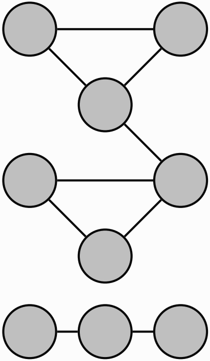
Problem: We have more vertices than real processors (workers)

Solution:

- Assign each vertex to a real processor (worker). E.g. by hash-partitioning, or some more clever form of partitioning.
- In each superstep, the real workers *accumulate* the messages sent by the vertices to other vertices, and communicate these to the corresponding workers.

- Idea first proposed by Google in the paper "Pregel: A System for Large-Scale Graph Processing", SIGMOD 2010. Also provides fault-tolerance mechanism.
- Open-source implementation by Apache Giraph; also supported in Spark GraphX.
- Commercial and very efficient implementation by GraphLab, later acquired by Apple.

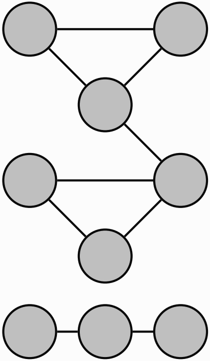
VERTEX-CENTRIC BSP



Vertex-Centric BSP:

- The majority of graph algorithms are iterative, and traverse the graph in some way.
- Vertex-centric BSP gives a natural way of expressing these algorithms in a parallel fashion by "thinking like a vertex".

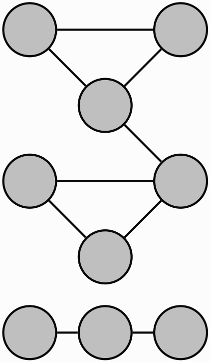
VERTEX-CENTRIC BSP



Vertex-Centric BSP:

- The majority of graph algorithms are iterative, and traverse the graph in some way.
- Vertex-centric BSP gives a natural way of expressing these algorithms in a parallel fashion by "thinking like a vertex".
- However, as we'll see in the next section, the obtained implementations need not be the most efficient ones!

ANOTHER EXAMPLE: CONNECTED COMPONENTS

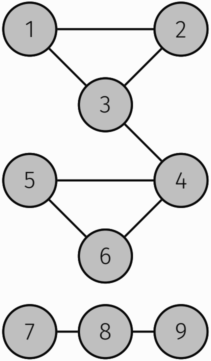


Connected Components in Vertex-BSP.

(For undirected graphs)

- Initially, each node has a distinct label (id)
- In each superstep, nodes communicate their label to neighbouring nodes, and keep the minimum of their current label, and the labels in received messages.
- Keep doing new supersteps until no label changes anymore.
- Upon convergence, each node in a connected component has the same label.

ANOTHER EXAMPLE: CONNECTED COMPONENTS

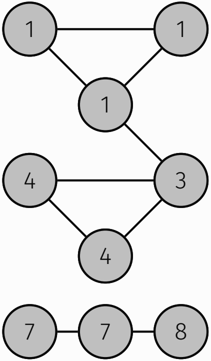


Initial IDs

Connected Components in Vertex-BSP.
(For undirected graphs)

- Initially, each node has a distinct label (id)
- In each superstep, nodes communicate their label to neighbouring nodes, and keep the minimum of their current label, and the labels in received messages.
- Keep doing new supersteps until no label changes anymore.
- Upon convergence, each node in a connected component has the same label.

ANOTHER EXAMPLE: CONNECTED COMPONENTS



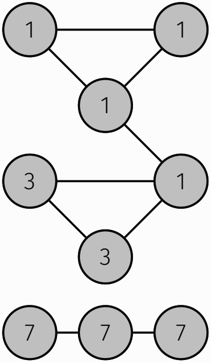
After superstep 1

Connected Components in Vertex-BSP.

(For undirected graphs)

- Initially, each node has a distinct label (id)
- In each superstep, nodes communicate their label to neighbouring nodes, and keep the minimum of their current label, and the labels in received messages.
- Keep doing new supersteps until no label changes anymore.
- Upon convergence, each node in a connected component has the same label.

ANOTHER EXAMPLE: CONNECTED COMPONENTS

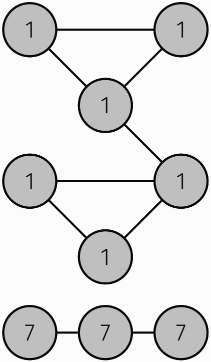


After superstep 2

Connected Components in Vertex-BSP.
(For undirected graphs)

- Initially, each node has a distinct label (id)
- In each superstep, nodes communicate their label to neighbouring nodes, and keep the minimum of their current label, and the labels in received messages.
- Keep doing new supersteps until no label changes anymore.
- Upon convergence, each node in a connected component has the same label.

ANOTHER EXAMPLE: CONNECTED COMPONENTS



After superstep 3

Connected Components in Vertex-BSP.
(For undirected graphs)

- Initially, each node has a distinct label (id)
- In each superstep, nodes communicate their label to neighbouring nodes, and keep the minimum of their current label, and the labels in received messages.
- Keep doing new supersteps until no label changes anymore.
- Upon convergence, each node in a connected component has the same label.

SPEEDUP AND SCALEUP

Parallelism example from Lecture 1

- "Embarassingly parallel" example: count the number of times the word "Belgium" appears in documents on the Web.

- Each server has multiple CPUs and can read from multiple disks in parallel. As such, each server can analyze many documents in parallel.
- At the end, sum the per-server counters (which can be done very fast)

Parallelism example from Lecture 1

- Let us consider the maximal aggregate bandwidth: the speed by which we can analyze data in parallel assuming ideal data distribution over servers & disks

Component		Max Aggr Bandwidth
1 Hard Disk		100 MB/sec (≈ 1 Gbps)
Server	= 12 Hard Disks	1.2 GB/sec (≈ 12 Gbps)
Rack	= 80 servers	96 GB/sec (≈ 768 Gbps)
Cluster/datacenter	= 30 racks	2.88 TB/sec (≈ 23 Tbps)

- Scanning 400TB hence takes 138 secs ≈ 2.3 minutes
- Scanning 400TB *sequentially* at 100 MB/sec takes ≈ 46.29 days

Parallelism example from Lecture 1

- Let us consider the maximal aggregate bandwidth: the speed by which we can analyze data in parallel assuming ideal data distribution over servers & disks

Component		Max Aggr Bandwidth
1 Hard Disk		100 MB/sec (≈ 1 Gbps)
Server	= 12 Hard Disks	1.2 GB/sec (≈ 12 Gbps)
Rack	= 80 servers	96 GB/sec (≈ 768 Gbps)
Cluster/datacenter	= 30 racks	2.88 TB/sec (≈ 23 Tbps)

- Scanning 400TB hence takes 138 secs ≈ 2.3 minutes
- Scanning 400TB *sequentially* at 100 MB/sec takes ≈ 46.29 days

- Parallelism hence gives a **speed-up** of 28 800 x

Parallelism example from Lecture 1

- Let us consider the maximal aggregate bandwidth: the speed by which we can analyze data in parallel assuming ideal data distribution over servers & disks

Component		Max Aggr Bandwidth
1 Hard Disk		100 MB/sec (≈ 1 Gbps)
Server	= 12 Hard Disks	1.2 GB/sec (≈ 12 Gbps)
Rack	= 80 servers	96 GB/sec (≈ 768 Gbps)
Cluster/datacenter	= 30 racks	2.88 TB/sec (≈ 23 Tbps)

- Scanning 400TB hence takes 138 secs ≈ 2.3 minutes
- Scanning 400TB *sequentially* at 100 MB/sec takes ≈ 46.29 days

- Parallelism hence gives a **speed-up** of 28 800 x
- It is not a coincidence that the number of parallel resources we are using is:

12 hard disks x 80 servers x 30 racks = 28 800

SPEEDUP

Definition

Speedup is the ratio of the latency of two systems, A and B, when run on the same problem (of the same size).

SPEEDUP

Definition

Speedup is the ratio of the latency of two systems, A and B, when run on the same problem (of the same size).

Example:

- A: Sequential scan of 400 TB with 1 hard drive takes $\approx$ 46.29 days
- B: Parallel scan of 400 TB with 28 800 hard drives takes $\approx$ 2.3 minutes

$$\text{Speedup of } B \text{ w.r.t. } A = \frac{A}{B} = 28800$$

SPEEDUP

Definition

Speedup is the ratio of the latency of two systems, A and B, when run on the same problem (of the same size).

Example:

- A: Sequential scan of 400 TB with 1 hard drive takes ≈ 46.29 days
- B: Parallel scan of 400 TB with 28 800 hard drives takes ≈ 2.3 minutes

$$\text{Speedup of } B \text{ w.r.t. } A = \frac{A}{B} = 28800$$

- This is linear speedup: adding 28800 times the resources gets the job done 28800 times quicker.

SPEEDUP

Definition

Speedup is the ratio of the latency of two systems, A and B, when run on the same problem (of the same size).

Example:

- A: Sequential scan of 400 TB with 1 hard drive takes $\approx$ 46.29 days
- B: Parallel scan of 400 TB with 28 800 hard drives takes $\approx$ 2.3 minutes

$$\text{Speedup of } B \text{ w.r.t. } A = \frac{A}{B} = 28800$$

- This is linear speedup: adding 28800 times the resources gets the job done 28800 times quicker.

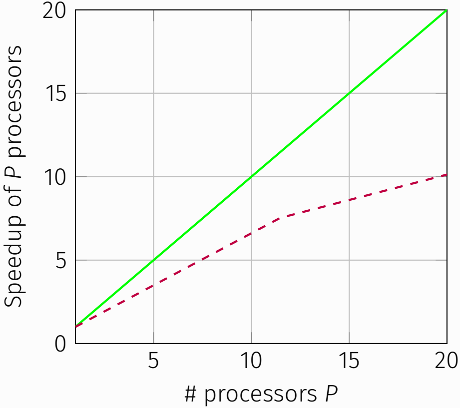
Bottom line:

- More processors $\rightarrow$ higher speed.
- Linear speedup ideal (but not always possible).

SPEEDUP

Definition

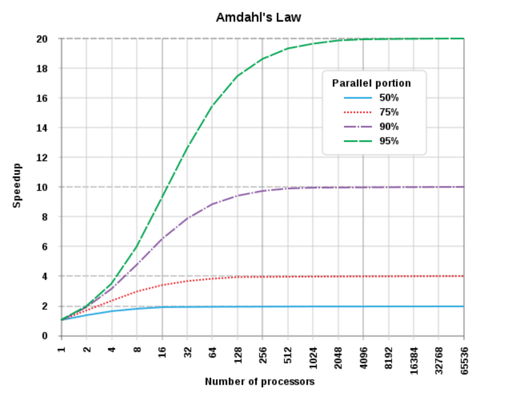
Speedup is the ratio of the latency of two systems, *A* and *B*, when run on the same problem (of the same size).



Linear vs non-linear speedup.

AMDAHL'S LAW

Amdahl's law is a formula that gives the theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved.



Amdahl's law:

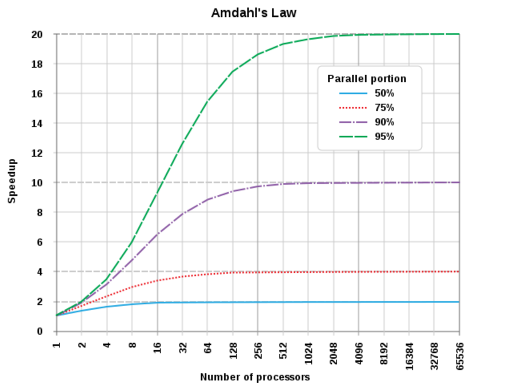
$$\text{Speedup}(P) = \frac{1}{(1-\alpha) + \frac{\alpha}{P}}$$

where

- P denotes the number of parallel processors
- α denotes the fraction of the task that benefits from parallelism;
- $1 - \alpha$ denotes the fraction of the task that runs inherently sequential.

AMDAHL'S LAW

Amdahl's law is a formula that gives the theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved.



Amdahl's law:

$$\text{Speedup}(P) = \frac{1}{(1-\alpha) + \frac{\alpha}{P}}$$

where

- P denotes the number of parallel processors
- α denotes the fraction of the task that benefits from parallelism;
- $1 - \alpha$ denotes the fraction of the task that runs inherently sequential.

Observe:

$$\lim_{P \to \infty} \text{Speedup}(P) = \frac{1}{(1-\alpha)}$$

SCALEUP

Definition

Scalability is the capacity of a system to handle a growing amount of work by adding a growing amount of resources.

Scaleup = is the ratio between the amount of work that two systems, A and B, process when running for the same amount of time.

SCALEUP

Definition

Scalability is the capacity of a system to handle a growing amount of work by adding a growing amount of resources.
Scaleup = is the ratio between the amount of work that two systems, A and B, process when running for the same amount of time.

Example:

- A: in 10 minutes, 10 servers process 100 TB of data.
- B: in 10 minutes, 22 servers process 200 TB of data.

$$\text{Scaleup of } B \text{ w.r.t. } A = \frac{200}{100} = 2$$

SCALEUP

Definition

Scalability is the capacity of a system to handle a growing amount of work by adding a growing amount of resources.
Scaleup = is the ratio between the amount of work that two systems, A and B, process when running for the same amount of time.

Example:

- A: in 10 minutes, 10 servers process 100 TB of data.
- B: in 10 minutes, 22 servers process 200 TB of data.

$$\text{Scaleup of } B \text{ w.r.t. } A = \frac{200}{100} = 2$$

- This is sub-linear scaleup: we require 2.2 times more resources to perform 2 times the amount of work.

SCALEUP

Definition

Scalability is the capacity of a system to handle a growing amount of work by adding a growing amount of resources.
Scaleup = is the ratio between the amount of work that two systems, A and B, process when running for the same amount of time.

Example:

- A: in 10 minutes, 10 servers process 100 TB of data.
- B: in 10 minutes, 22 servers process 200 TB of data.

$$\text{Scaleup of } B \text{ w.r.t. } A = \frac{200}{100} = 2$$

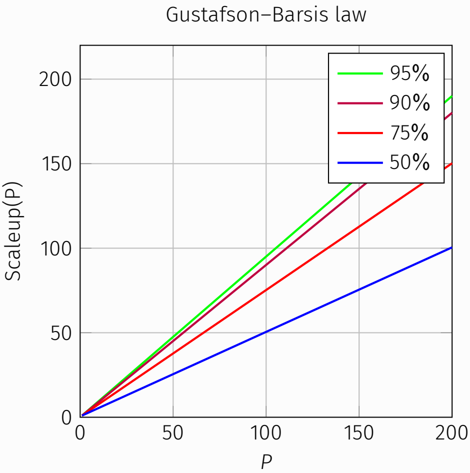
- This is sub-linear scaleup: we require 2.2 times more resources to perform 2 times the amount of work.

Bottom line:

- More processors $\rightarrow$ can process more data (in the same time)
- Linear scaleup ideal (but not always possible).

GUSTAFSON–BARSIS'S LAW

The Gustafson-Barsis law is a formula that gives the theoretical scaleup that can be expected of a system whose resources are improved.



Gustafson–Barsis's law:

$$\text{Scaleup}(P) = (1 - \alpha) + \alpha \cdot P$$

where

- P denotes the number of parallel processors
- α denotes the fraction of the task that benefits from parallelism;
- $1 - \alpha$ denotes the fraction of the task that runs inherently sequential.

EMBARASSING PARALLELISM: A DEFINITION

> “*In parallel computing, an embarrassingly parallel workload or problem (also called perfectly parallel or pleasingly parallel) is one where little or no effort is needed to separate the problem into a number of parallel tasks.*
>
> *This is often the case where there is little or no dependency or need for communication between those parallel tasks, or for results between them.*

—Wikipedia

EMBARASSING PARALLELISM: A DEFINITION

> "*In parallel computing, an embarrassingly parallel workload or problem (also called perfectly parallel or pleasingly parallel) is one where little or no effort is needed to separate the problem into a number of parallel tasks.*
>
> *This is often the case where there is little or no dependency or need for communication between those parallel tasks, or for results between them.*

—Wikipedia

In other words:
An embarrasingly parallel problem is a problem where $\alpha \approx 100\%$, i.e., there is little or no inherent sequential computation involved (e.g.. "count occurrences of Belgium").

EMBARASSING PARALLELISM: A DEFINITION

> "*In parallel computing, an embarrassingly parallel workload or problem (also called perfectly parallel or pleasingly parallel) is one where little or no effort is needed to separate the problem into a number of parallel tasks.*
>
> *This is often the case where there is little or no dependency or need for communication between those parallel tasks, or for results between them.*

—Wikipedia

In other words:
An embarrasingly parallel problem is a problem where $\alpha \approx 100\%$, i.e., there is little or no inherent sequential computation involved (e.g.. "count occurrences of Belgium").

In other words:

- When phrased as a BSP algorithm, the number of supersteps S is fixed (does not depend on problem size), and the total work W and total communication H are inversely proportional to the number of processors P.

OTHER PROBLEMS

Problems on graphs are typical examples of problems where parallelism can help, but which are not embarassingly parallel.

SCALABILITY, BUT AT WHAT COST?

The norm for data analytics is now to run them on commodity clusters with programming frameworks such as M/R and Spark …

—Rowstron et al., HotCDP 2012

The norm for data analytics is now to run them on commodity clusters with programming frameworks such as M/R and Spark ...

...we believe that we could now say that "nobody ever got fired for using Hadoop on a cluster"!

—Rowstron et al., HotCDP 2012

A WORD OF CAUTION

- Running Hadoop/Spark is regarded as the cool thing to do.
- Clusters-as-a service has become extremely easy.

A WORD OF CAUTION

- Running Hadoop/Spark is regarded as the cool thing to do.
- Clusters-as-a service has become extremely easy.

Cost-benefit analysis required.

- But running computation on a rented cluster costs money.
- Nobody likes their AWS/Azure/GCP bill.
- Did you really need this infrastructure for your computation?

A WORD OF CAUTION

- Running Hadoop/Spark is regarded as the cool thing to do.
- Clusters-as-a service has become extremely easy.

Cost-benefit analysis required.

- But running computation on a rented cluster costs money.
- Nobody likes their AWS/Azure/GCP bill.
- Did you really need this infrastructure for your computation?

Case in point: graph analytics (next).

QUIZ

Speed-up of a data-parallel algorithm before (A) and after (B) a change was made to the algorithm. Which system would you prefer?

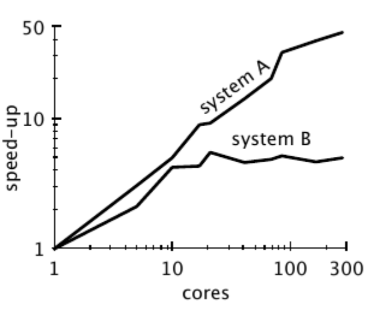


Figure source: McSherry et al. Scalability! But at what COST? HOTOS workshop, 2015.

SPEED-UP ISN'T EVERYTHING

Speed-up and total runtime of a data-parallel algorithm before (A) and after (B) a change was made to the algorithm. Which system would you prefer?

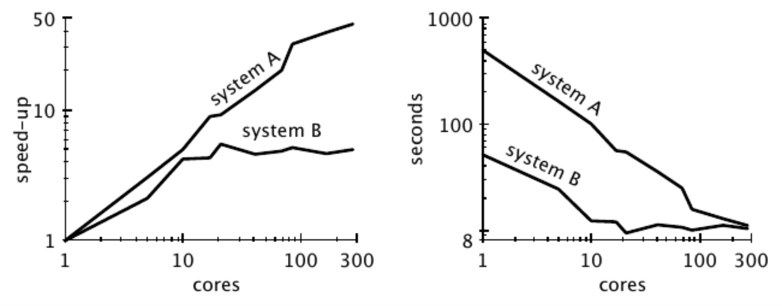
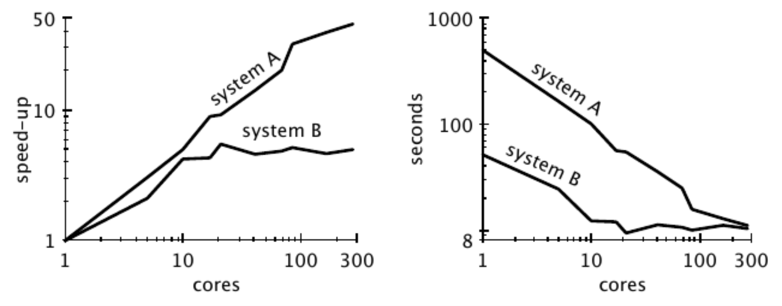


Figure source: McSherry et al. Scalability! But at what COST? HOTOS workshop, 2015.

SPEED-UP ISN'T EVERYTHING

Speed-up and total runtime of a data-parallel algorithm before (A) and after (B) a change was made to the algorithm. Which system would you prefer?



Bottom line:

Speedup and scaleup don't mean anything if they are because of system inefficiencies (overheads) that are parallelizable.

Figure source: McSherry et al. Scalability! But at what COST? HOTOS workshop, 2015.

PARALLELISM ISN'T EVERYTHING

name	twitter_rv [13]	uk-2007-05 [5, 6]
nodes	41,652,230	105,896,555
edges	1,468,365,182	3,738,733,648
size	5.76GB	14.72GB

Two graph datasets

From: McSherry et al. Scalability! But at what COST? HOTOS workshop, 2015.

PARALLELISM ISN'T EVERYTHING

name	twitter_rv [13]	uk-2007-05 [5, 6]
nodes	41,652,230	105,896,555
edges	1,468,365,182	3,738,733,648
size	5.76GB	14.72GB

Two graph datasets

scalable system	cores	twitter	uk-2007-05
GraphChi [12]	2	3160s	6972s
Stratosphere [8]	16	2250s	-
X-Stream [21]	16	1488s	-
Spark [10]	128	857s	1759s
Giraph [10]	128	596s	1235s
GraphLab [10]	128	249s	833s
GraphX [10]	128	419s	462s
Single thread (SSD)	1	300s	651s
Single thread (RAM)	1	275s	-

Reported elapsed times for 20 PageRank iterations compared to a single-threaded implementation.

From: McSherry et al. Scalability! But at what COST? HOTOS workshop, 2015.

PARALLELISM ISN'T EVERYTHING

name	twitter_rv [13]	uk-2007-05 [5, 6]
nodes	41,652,230	105,896,555
edges	1,468,365,182	3,738,733,648
size	5.76GB	14.72GB

Two graph datasets

scalable system	cores	twitter	uk-2007-05
GraphLab	128	249s	833s
GraphX	128	419s	462s
Vertex order (SSD)	1	300s	651s
Vertex order (RAM)	1	275s	-
Hilbert order (SSD)	1	242s	256s
Hilbert order (RAM)	1	110s	-

Reported elapsed times for 20 PageRank iterations compared to an improved single-threaded implementation (traversing edges in different order).

From: McSherry et al. Scalability! But at what COST? HOTOS workshop, 2015.

PARALLELISM ISN'T EVERYTHING

name	twitter_rv [13]	uk-2007-05 [5, 6]
nodes	41,652,230	105,896,555
edges	1,468,365,182	3,738,733,648
size	5.76GB	14.72GB

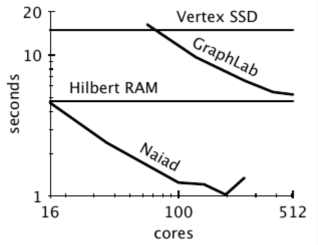
Two graph datasets

scalable system	cores	twitter	uk-2007-05
Stratosphere [8]	16	950s	-
X-Stream [21]	16	1159s	-
Spark [10]	128	1784s	$\geq$ 8000s
Giraph [10]	128	200s	$\geq$ 8000s
GraphLab [10]	128	242s	714s
GraphX [10]	128	251s	800s
Single thread (SSD)	1	153s	417s

Reported elapsed times for connected components (based on label propagation) compared to a single-threaded implementation (not using label propagation).

From: McSherry et al. Scalability! But at what COST? HOTOS workshop, 2015.

COST



Definition

Define the COST of a system (on a given workload/dataset) to be the Configuration at which it Outperforms a Single Threaded optimized algorithm.

- The COST hence quantifies when it becomes useful to use a distributed processing system for a given workload
- If the COST is high, a Single-Threaded implementation may actually be more preferable from an economic viewpoint.
- Some systems have unbounded COST on certain graph-related problems.

Check out the full talk about the COST paper at:
`https://www.youtube.com/watch?v=6bWBEJBMNG0`

IN CONCLUSION

- Big-data programming frameworks such as M/R and Spark can allow analysis of huge datasets.
- These frameworks introduce their own overheads; avoid network communication if possible.
- The BSP model is natural theoretical model for the formulation and analysis of distributed parallel algorithms.
- There are limits to speedup; scaleup is more favorable.
- Big-data programming framworks are great for “embarassingly parallel” problems.
- For problems that are more difficult to parallelize (e.g. graph problems), it may not always make sense to use distributed processing. A good central algorithm can go a long way.

REFERENCES

- Rob H. Bisseling. *Parallel Scientific Computation: A Structured Approach using BSP and MPI*, Oxford University Press, March 2004.
- Peter Zecević and Marko Bonaći. *Spark in Action*, Manning, 2017.
- Leslie Valiant. *A bridging model for parallel computation*, Communications of the ACM, 33(8), 1990
- Malewicz et al. *Pregel: a system for large-scale graph processing*, SIGMOD conference, 2010.
- Frank McSherry, Michael Isard, Derek G. Murray. *Scalability! But at what COST?*. HOTOS workshop, 2015.

QUESTIONS?

ACKNOWLEDGEMENTS

Based on content created by Stijn Vansummeren.