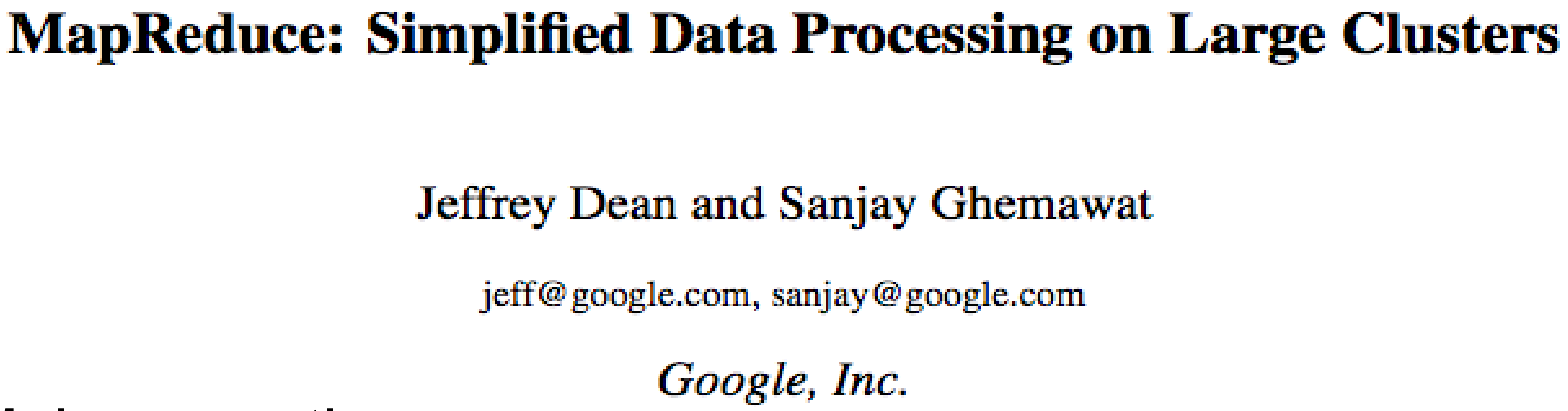
Distributed Data-Parallel Programming

Distributed Data-Parallel Programming

- We have studied parallel collections
  - parallel tasks running in the same machine sharing the same memory
  - we have seen the impact on programming of concurrent access to memory
    - Bottleneck, data corruption, ...
- We now move to a distributed scenario
  - Data can be partitioned and distributed on different machines (that do not share memory)
  - Two main new concerns:
    - Partial failures: one node crashes, but the entire computation should not
    - Network latency: inter-node interaction is orders of magnitude slower

Google MapReduce

- In early 2000, Google proposed a novel programming model (supported by a corresponding framework) for data processing on large clusters:



MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

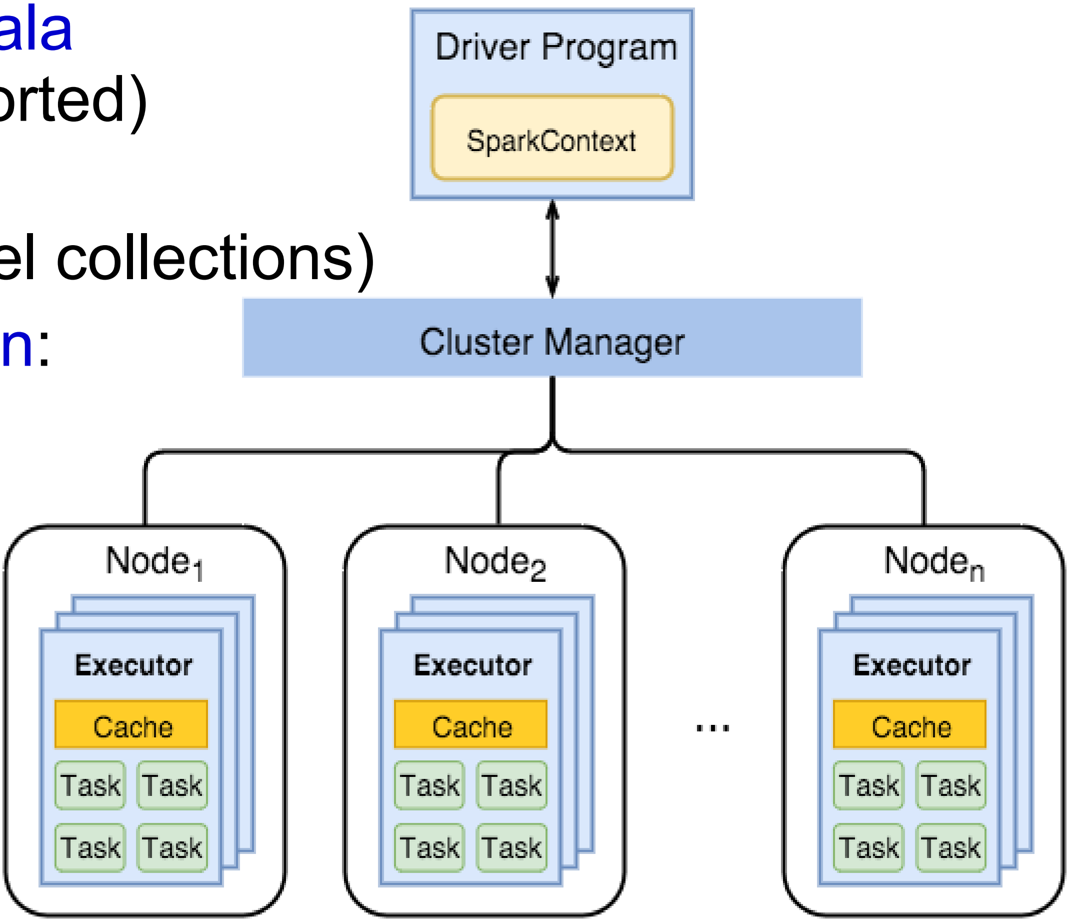
  - Main properties:
    - The framework exposes an easy to use interface (programs are expressed as map-reduce steps)
    - Fault tolerant
  - Apache Hadoop is an open-source implementation of a map-reduce framework

Pros-Cons of Hadoop

- Fault tolerance permitted to run big-data applications on thousands of nodes
  - probability of one-node failure is not negligible
- But, fault tolerance comes at a price:
  - Between each map-reduce, data are always stored on disk
  - In case of failure, they can be restored
- Disk storage have a negative impact on performances
  - Apache Spark significantly improves Hadoop:
    - Immutable datasets
    - Intermediary data stored in memory
    - Modifications to the datasets are recomputed in case of failure

Spark

- Spark makes distributed data parallel programming more efficient and easy:
  - Natural to be used within Scala (Python and Java also supported)
  - Higher-order programming (as for collections and parallel collections)
  - It provides the RDD collection: Resilient Distributed Dataset
  - Data in an RDD are distributed over nodes:
    - The programmer refers to the entire distributed dataset as it was a unique collection



How to create an RDD

- New RDDs are created by exploiting a `SparkContext` object:
  - It is the handle to the Spark cluster (connection between the cluster and the running application)
  - Two main methods:
    - `parallelize`: convert a local collection to an RDD
    - `textFile`: read a text file, return an `RDD[String]` (one String per line)
- Another way to create an RDD is:
  - Apply higher-order functions called transformations (similar to map or filter) to an existing RDDs

```
val largelist: List[String] = ...
val wordsRdd = sc.parallelize(largelist)
val lengthsRdd = wordsRdd.map(_.length)
```

A first example

- WordCount is considered the HelloWorld for distributed data parallel programming
  - This is the Scala+Spark version:

```
val input = sc.textFile(inputFile)
val counts = input.flatMap(line => line.split(" ")).
  map(w => (w.filter(_.isLetter).toUpperCase, 1)).
  reduceByKey((x, y) => x + y)
```

  - Following the programming model initiated by Google MapReduce, it is based on a map-reduce computation
    - The data elaboration is executed in parallel on each data partition, and only the local reduce results are communicated and combined

Higher-order functions on RDD

- The functions available for computations on RDDs are classified in:
  - Transformations: return a new RDD
    - Examples are `map`, `flatMap`, `filter` and `distinct`
  - Actions: do not return an RDD
    - Examples are `collect`, `count`, `take`, `reduce`, `foreach`, `takeSample`, `takeOrdered` and `saveAsTextFile`
  - There are also transformations on two RDDs:
    - Examples are `union`, `intersection`, `subtract` and `cartesian`

IMPORTANT NOTE:
Transformations are lazy, i.e., evaluated only when necessary to compute a subsequent action

Transformations signature

`map[B](f: A=> B): RDD[B]`

`flatMap[B](f: A=> Traversable[B]): RDD[B]`

`filter(pred: A=> Boolean): RDD[A]`

`distinct(): RDD[B]` (remove duplicates)

- Transformations on two RDDs

`union(other: RDD[T]): RDD[T]`
(do not remove duplicates)

`intersection(other: RDD[T]): RDD[T]`
(remove duplicates)

`subtract(other: RDD[T]): RDD[T]`

`cartesian[U](other: RDD[U]): RDD[(T, U)]`

Actions signature

```
collect(): Array[T]
count(): Long
take(num: Int): Array[T]
reduce(op: (A, A) => A): A
fold(z: A)(f: (A, A)=> A): A
aggregate[B](z: B)(seqop: (B, A)=> B,
       combop: (B, B) => B): B
foreach(f: T => Unit): Unit
takeSample(withRepl: Boolean, num: Int): Array[T]
takeOrdered(num: Int)(
       implicit ord: Ordering[T]): Array[T]
saveAsTextFile(path: String): Unit
```

Lazy evaluation of transformations

- Laziness of transformations is at the basis of the improved Spark performances:
  - Spark do not compute immediately the data resulting from transformations
  - Deferring transformations allows Spark to analyse and optimize the chain of operations before executing it:
    - All transformations are stored in a DAG (direct acyclic graph):
      - Transformations are organized in stages
      - Each stage contains a sequence of "narrow" transformations (like `map` or `filter`)
      - "wide" transformations (like `intersection`) start new stages
      - Narrow transformations can be executed locally in each partition
      - Wide transformations needs shuffling (inter-partition data exchange); these represent stage boundaries
- Evaluation of transformations are triggered by actions

Pros of Laziness

- Deferring the evaluation of transformations allows Spark:
  - to compute only what is needed
    - If an action takes only few data, only those strictly needed are computed

```
val lastYearslogs: RDD[String] = ...
val firstlogsWithErrors =
lastYearslogs.filter(_.contains("ERROR")).take(10)
```

  - to minimize data traversal
    - Narrow transformations can be pipelined to be computed in one pass

```
time(words.filter(w => w.length>10).count())
val pairs = words.map(w => w.length)
val longWords = pairs.filter(l => (l > 10))
time(longWords.count())
```

Cons of Laziness

- Transformations could be computed several times:
  - If more than one action depends on the same transformation, it is re-computed for each action
  - This problem can be avoided by asking to make the intermediary data persistent

```
val counts = words.
  map(word =>
      (word.filter(_.isLetter).toUpperCase, 1)).
  reduceByKey((x, y) => x + y).persist()
counts.saveAsTextFile(outputFile)
counts.reduce(( p1: (String,Int),
                p2:(String,Int)) =>
                if (p1._2 > p2._2) p1 else p2))
```

Persistence

- Persistence can be customized by passing to the `persist()` method the storage level (add _n for n replications):

Level	*Space used*	*CPU time*	*In memory*	*On disk*
MEMORY_ONLY	High	Low	Y	N
MEMORY_ONLY_SER	Low	High	Y	N
MEMORY_AND_DISK*	High	Medium	Some	Some
MEMORY_AND_DISK_SER†	Low	High	Some	Some
DISK_ONLY	Low	High	N	Y

```
import org.apache.spark.storage.StorageLevel
val counts = words.
  map(word =>
    (word.filter(_.isLetter).toUpperCase, 1)).
  reduceByKey((x, y) => x + y).
  persist(StorageLevel.DISK_ONLY_2)
```

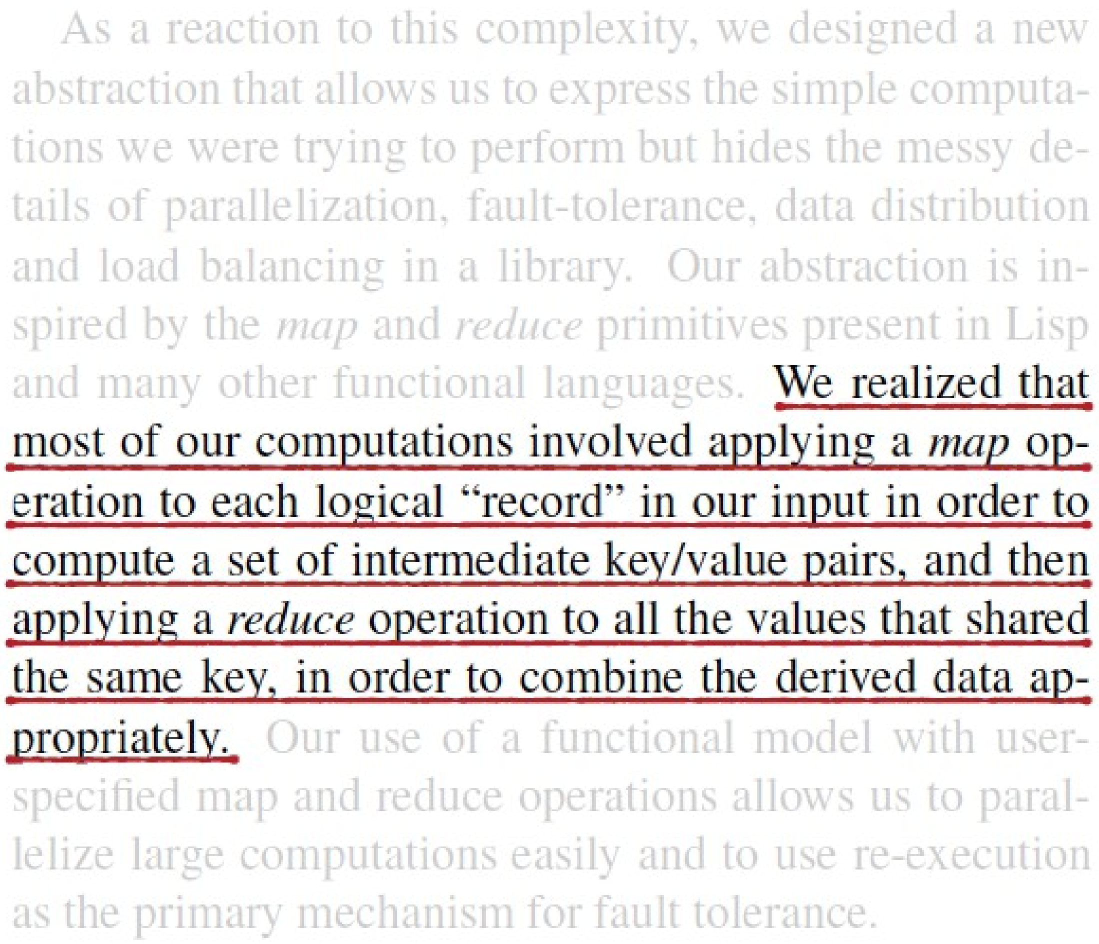
Application vs Executor Evaluation

- Tranformations are evaluated by executors inside the computing nodes
  - Narrow locally, wide with shuffling
- Actions return results to application through the driver

```
case class Person (name: String age: Int)
val people: RDD[Person] =  ...
people.foreach(println) //executed by the executor
people.collect().foreach(println)//by the application
```

Key-Value Pairs

- The programming model initiated by MapReduce used key-value pairs as basic way to store intermediary data:



As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. 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. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

Key-Value RDDs

- Key-Value pairs are present also in Spark:
  - An RDD[(K,V)] is a collection of key (of type K) - value (of type V) pairs
    - Useful for parallel computation on distinct keys and for regrouping data (i.e. all pairs with the same key located in the same partition)
  - Additional transformations/actions for key-value RDDs:

```
groupByKey(): RDD[(K, Iterable[V])
reduceByKey(func: (V,V)=> V): RDD[(K, V)]
join[W]: (other: RDD[(K, W)]: RDD[(K, (V, W))]
leftOuterJoin[W](other: RDD[(K, W)]):
     RDD[(K, (V, Option[W]))]
rightOuterJoin[W](other: RDD[(K, W)]):
     RDD[(K, (Option[V], W))]
```

Shuffling

- Operations like `groupByKey` group all pairs with the same key in a unique new pair
  - Moving pairs to the same node is necessary: this movement is called shuffling
  - Shuffling is the most expensive activity
- The programmer should minimize shuffling being aware of the shuffling performed by the functions:
  - E.g. `reduceByKey` computes local intermediary data and communicates across the network only such data

Example

- Compute the number and the total amount of the purchases of each customer:

```scala
case class Purchase(customerId: Int, price: Double)
val purchasesRdd:org.apache.spark.rdd.RDD[Purchase] =
  sc.textFile(inputFile)

val purchasesPerCustomer: Array[(Int,(Int,Double))] =
  purchasesRdd.
  map(p => (p.customerId, p.price)).   // Pair RDD
  groupByKey().    // returns RDD[(K,Iterable[V])]
  map(p => (p._1, (p._2.size, p._2.sum))).
  collect()
```

Example

- Compute the number and the total amount of the purchases of each customer:

```
case class Purchase(customerId: Int, price: Double)
val purchasesRdd:org.apache.spark.rdd.RDD[Purchase] =
  sc.textFile(inputFile)

val purchasesPerCustomer: Array[(Int,(Int,Double))] =
  purchasesRdd.
  map(p => (p.customerId, p.price)).   // Pair RDD
  groupByKey().    // returns RDD[(K,Iterable[V])]
  map(p => (p._1, (p._2.size, p._2.sum))).
  collect()
```

  - This version is not efficient because it shuffles all ill-placed pairs

Example

- Compute the number and the total amount of the purchases of each customer:

```
case class Purchase(customerId: Int, price: Double)
val purchasesRdd:org.apache.spark.rdd.RDD[Purchase] =
  sc.textFile(inputFile)

val purchasesPerCustomer: Array[(Int,(Int,Double))] =
  purchasesRdd.
  map(p => (p.customerId, (1, p.price))). //Pair RDD
  reduceByKey((v1, v2) =>
              (v1._1 + v2._1, v1._2 + v2._2)).
  collect()
```

  - More efficient: shuffles at most one pair per key from each node

Partitioning

- In pair RDDs, it is possible to customize distribution of pairs over the partitions as a function on the keys
  - Two available library functions:
    - `RangePartitioner`: partitions computed dividing the key domain (from minimum to maximum) in a given number of equally sized ranges
    - `HashPartitioner`: applies an hash function to keys and distributes according to the hash modulo the number of partitions

```
val purchasesPairs = purchasesRdd.
  map(p => (p.customerId, (1, p.price)))
val purchasesRangeP = purchasesPairs.
  partitionBy(new RangePartitioner(numPartitions,
                                   purchasesPairs))
val purchasesHashP = purchasesPairs.
  partitionBy(new HashPartitioner(numPartitions))
```

Example

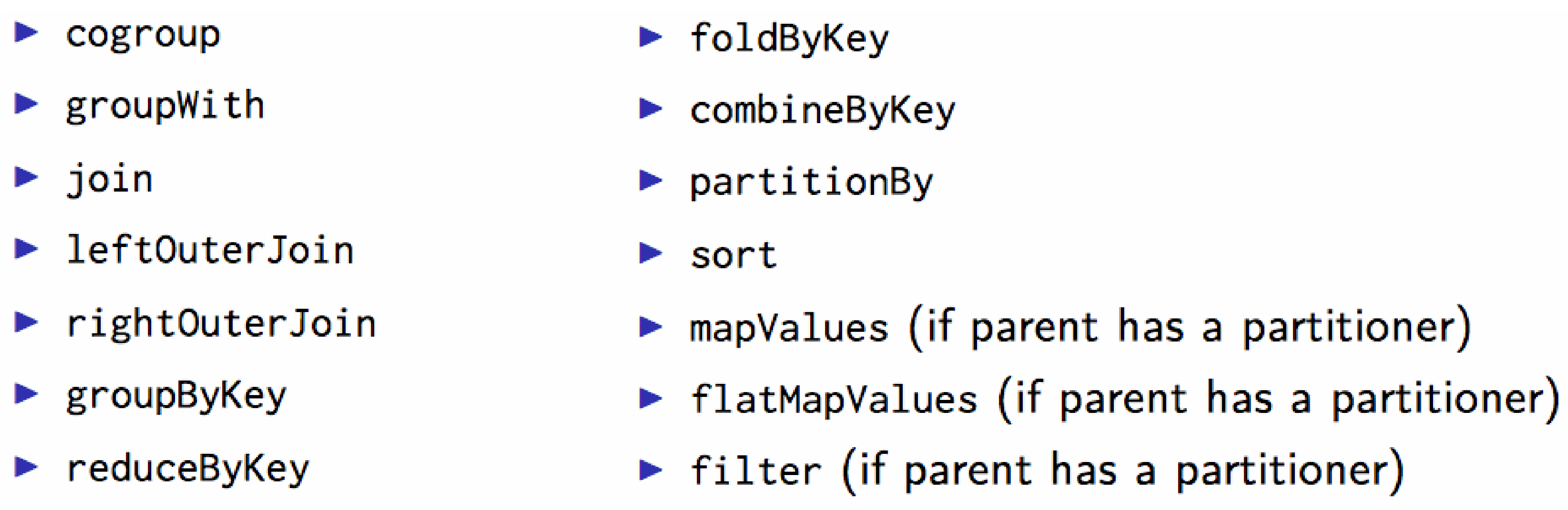
- Using a partitioner in the customer example, no shuffling occurs:

```
case class Purchase(customerId: Int, price: Double)
val purchasesRdd:org.apache.spark.rdd.RDD[Purchase] =
  sc.textFile(inputFile)

val purchasesPairs = purchasesRdd.
  map(p => (p.customerId, (1, p.price)))
val purchasesPartitioned = purchasesPairs.
  partitionBy(new RangePartitioner(numPartitions,
                                   purchasesPairs))
val purchases = purchasesPartitioned.
  reduceByKey((v1, v2) =>
              (v1._1 + v2._1, v1._2 + v2._2)).
  collect()
```

Implicit use of partitioner

- There are functions that hold (or propagate) a partitioner:



- cogroup
- groupWith
- join
- leftOuterJoin
- rightOuterJoin
- groupByKey
- reduceByKey
- foldByKey
- combineByKey
- partitionBy
- sort
- mapValues (if parent has a partitioner)
- flatMapValues (if parent has a partitioner)
- filter (if parent has a partitioner)

  – E.g. `mapValues` is preferable to `map` in case of partitioned RDDs because the partitioner is preserved

Example: PageRank

- Famous algorithm (named after Larry Page, co-founder of Google) used to rank importance of websites
- Consider a graph $G=(V,E)$, with arcs $(v,k) \in E$ meaning that document v has a link to document k
- Iterative algorithm that associates to each document k a rank $rank(k)$
  - In the first iteration $rank_0(k) = 1.0$, for all k
  - At each iteration, $rank_{i+1}(k) = 0.15+0.85*c_i(k)$, where

    $c_i(k) = \sum_{(v,k) \in E} rank_i(v) \ / \ |\{ t \mid (v,t) \in E \}|$

PageRank implementation

```
val links = edges.partitionBy(new HashPartitioner(4)).
            groupByKey().persist()
var ranks = links.mapValues(v => 1.0)

for(i <- 0 until 10) {
  val contributions = links.join(ranks).flatMap {
    case (u, (uLinks, urank)) =>
      uLinks.map(t => (t, urank / uLinks.size))
    }
  ranks = contributions.
     reduceByKey((x,y) => x+y).
     mapValues(v => 0.15+0.85*v)
}

ranks.saveAsTextFile(outputFile)
```

Scalability

- How much faster can a given problem be solved with *n* nodes instead of one?
- How much more work can be done with *n* nodes instead of one?
- What impact for the communication requirements of the distributed application have on performance?
- What fraction of the resources is actually used productively for solving the problem?

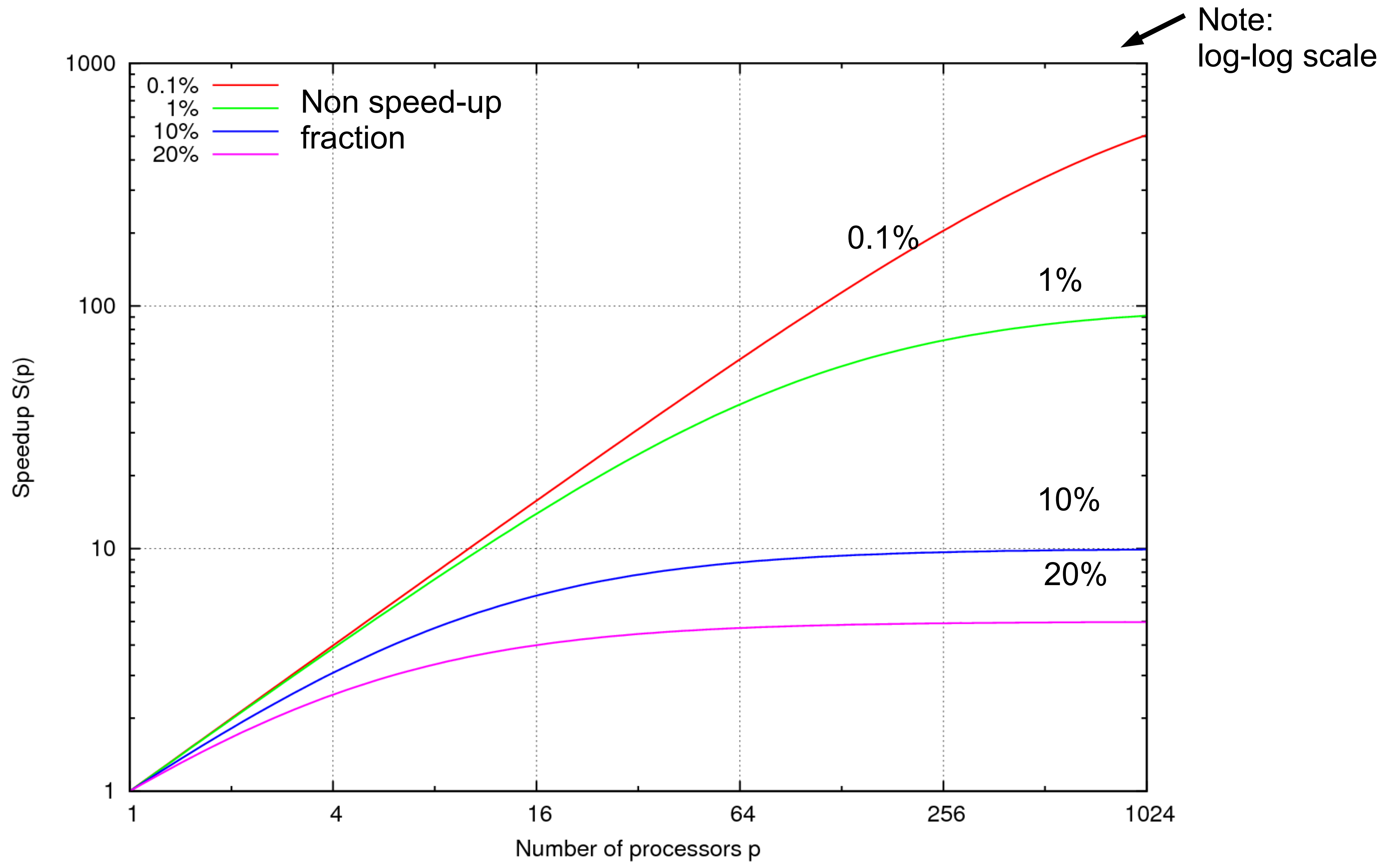
Speedup

- Let us define:
  - *n* = Number of nodes
  - *T*(*n*) = Execution time of the parallel program with *n* nodes
- Speedup *S*(*n*)

$$S(n) = \frac{T(1)}{T(n)}$$

- In the ideal case, the program with *n* nodes requires *1/n* the time of the program with *1* node
- *S*(*n*) = *n* is the ideal case of linear speedup
  - Realistically, $S(n) \le n$

Speedup (according to Amdahl's law)



Scaling Efficiency

- *Objective:*
  - Evaluate the impact of Amdahl's law on your distributed program
  - Quantify the effect on the execution time for each node that is added
- *Solution*: measure *Strong Scaling Efficiency*
  - Increase the number of nodes *n* keeping the *total* problem size fixed
  - The total amount of work remains constant, while the amount of work for each processor decreases as *n* increases
  - How to quantify the impact of each added node?
    - Divide the speedup for the number of nodes

Strong Scaling Efficiency

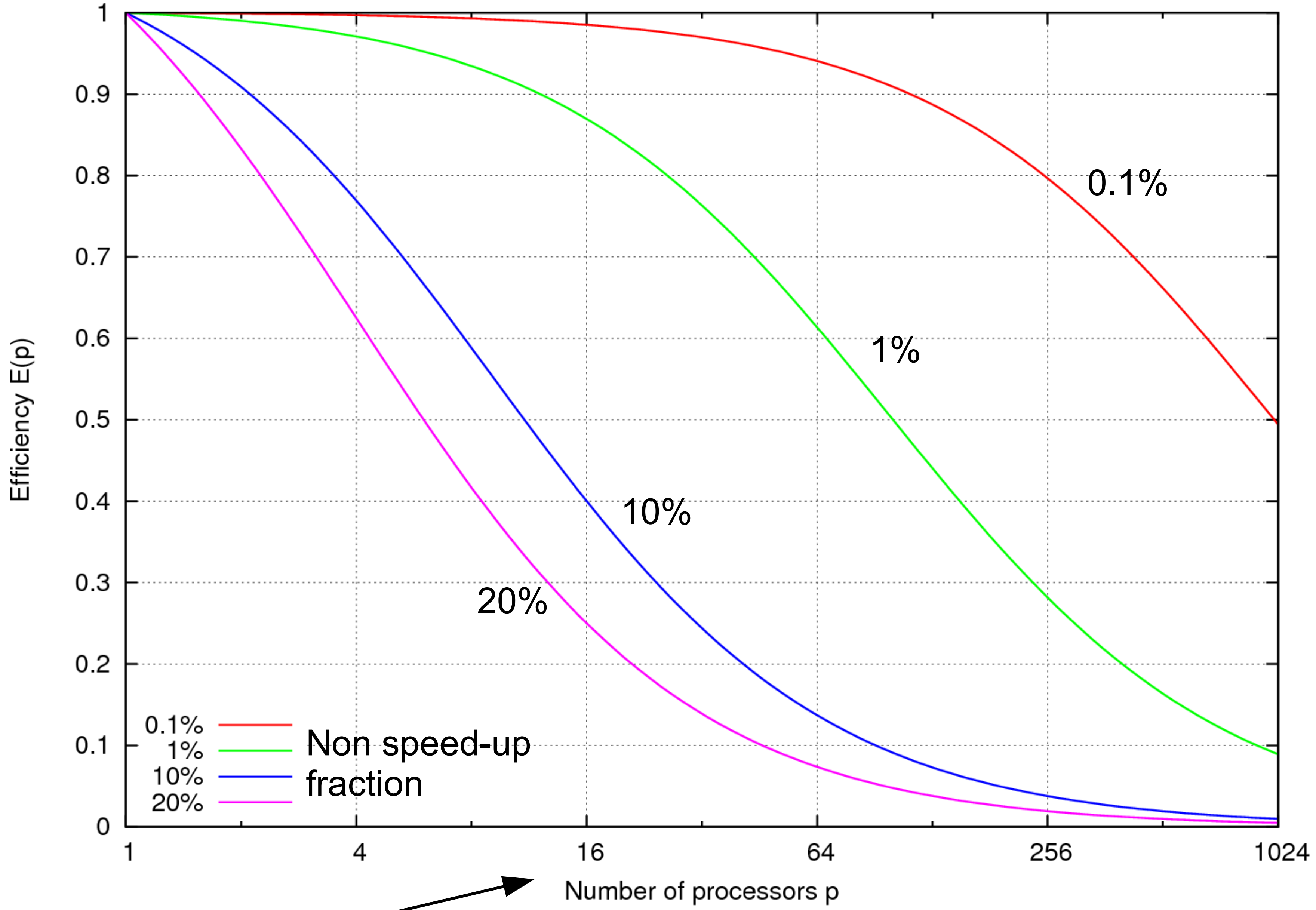
- $E(n)$ = Strong Scaling Efficiency

$$E(n) = \frac{S(n)}{n} = \frac{T(1)}{nT(n)}$$

where

- $T(n)$ = Execution time of the execution with n nodes

Strong Scaling Efficiency and Amdahl's Law



Note: logarithmic *x* scale

Negative result on strong scaling efficiency

- The strong scaling efficiency always tends to zero because the speedup is limited by the constant 1/f
- But in many cases we want to add computing resources to do more computational work instead of reducing the execution time for a fixed input
  - The goal is to solve larger problems within the same amount of time by increasing the computational power accordingly
  - How can we check whether adding computational power is actually effective to achieve this goal?

Weak Scaling Efficiency

- An alternative measure that considers increasing problem size is *Weak Scaling:*
  - Increase the number of nodes *n* keeping the per-node work fixed
  - The total amount of work grows as *n* increases
- *W*(n) = Weak Scaling Efficiency

$$W(n) = \frac{T_1}{T_n}$$

where

  - T_1 = time required to complete 1 work unit with 1 node
  - T_n = time required to complete *n* work units with *n* nodes
  - Weak scaling efficiency = 1 means that our program uses productively 100% of the added resources

Weak Scaling Efficiency

- Question: given the size of the input for 1 node (1 work unit) how much the input should be increased for n nodes (n work units)?
  - Let m be the input for 1 node: 1 work unit = $W(m)$ where $W(m)$ is the work complexity
  - Consider now n nodes: n work units = $n\ W(m)$ we need to find k such that $W(k\ m) = n\ W(m)$
- Examples:
  - Linear work complexity: $W(m) = m$
    $W(k\ m) = n\ W(m)$ implies $k\ m = n\ m$ hence $k = n$
  - Quadratic work complexity: $W(m) = m^2$
    $W(k\ m) = n\ W(m)$ implies $k^2\ m^2 = n\ m^2$ hence $k = n^{1/2}$
  - Cubic work complexity: $W(m) = m^3$
    $W(k\ m) = n\ W(m)$ implies $k^3\ m^3 = n\ m^3$ hence $k = n^{1/3}$