Computational Frameworks

MapReduce
OUTLINE

1. MapReduce
2. Basic Techniques and Primitives
MapReduce
Motivating scenario

At the dawn of the big-data era (early 2000’s), the following scenario was rapidly emerging

- In a wide spectrum of domains there was an increasing need to analyze large amounts of data.
- Available tools and commonly used platforms could not practically handle very large datasets.
  - Algorithms with high polynomial complexities were unfeasible, and platforms had limited main memory.
  - In extreme cases, even touching all data items would prove quite time consuming. (E.g., \( \approx 50 \cdot 10^9 \) web pages of about 20KB each \( \rightarrow 1000\)TB of data.)
- The use of powerful computing systems, which has been confined until then to a limited number of applications –typically from computational sciences (physics, biology, weather forecast, simulations)– was becoming necessary for a much wider array of applications.
Motivating scenario

The use of powerful computing systems based on either tightly-coupled parallel architectures or more loosely-coupled distributed platforms poses several problems

- These systems are costly to buy and to maintain, and they become rapidly obsolete.
- Fault-tolerance becomes serious issue: a large number of components implies a low *Mean-Time Between Failures* (MTBF).
- Developing software that effectively benefits from parallelism requires *sophisticated programming skills*. 
MapReduce

• Introduced by Google in 2004 as a programming framework for big data processing on distributed platforms.

• Its original formulation (see [DG08]) contained
  • A programming model
  • An implementation tailored towards a cluster-based computing environment.

• Since then, MapReduce implementations have been employed on clusters of commodity processors and cloud infrastructures for a wide array of big-data applications

• Main features:
  • Data centric view
  • Inspired by functional programming (map, reduce functions)
  • Ease of algorithm/program development. Messy details (e.g., task allocation; data distribution; fault-tolerance; load-balancing) are hidden to the programmer
Platforms

Typically, MapReduce applications run on

- **Clusters of commodity processors** (On-premises)
- **Platforms from cloud providers** (Amazon AWS, Microsoft Azure)
  - **IaaS** (*Infrastructure as a Service*): provides the users computing infrastructure and physical (or virtual) machines. This is the case of **CloudVeneto** which provides us a cluster of virtual machines.
  - **PaaS** (*Platform as a Service*): provides the users computing platforms with OS; execution environments, etc.
Platforms
Typical cluster architecture

- **Racks of 16-64 compute nodes** (commodity hardware), connected (within each rack and among racks) by fast switches (e.g., 10 Gbps Ethernet)

- **Distributed File System**
  - Files divided into *chunks* (e.g., 64MB per chunk)
  - Each chunk replicated (e.g., 2x or 3x) with replicas in different nodes and, possibly, in different racks
  - The distribution of the chunks of a file is represented into a *master node* file which is also replicated. A directory (also replicated) records where all master nodes are.
  - Examples: Google File System (GFS); **Hadoop Distributed File System** (HDFS)
Several software frameworks have been proposed to support MapReduce programming. For example:

- **Apache Hadoop**: most popular MapReduce implementation (*often used as synonymous of MapReduce*) from which an entire ecosystem of alternatives has stemmed, aimed at improving its (initially) very poor performance. The Hadoop Distributed File System is still widely used.

- **Apache Spark** (*learned in this course*): one of the most popular and widely used frameworks for big-data applications. It supports the implementation of MapReduce computations, but provides a much richer programming environment. It uses the HDFS.

**Important**: in this course we use the MapReduce programming model for the design and analysis of high-level algorithmic strategies, and Spark for their efficient implementation.
MapReduce computation

- Computation viewed as a sequence of rounds. *(The original formulation considered only one round.)*

- Each round transforms a set of key-value pairs into another set of key-value pairs *(data centric view!)*, through the following two phases

  - **Map phase:** a user-specified map function is applied separately to each input key-value pair and produces $\geq 0$ other key-value pairs, referred to as intermediate key-value pairs.

  - **Reduce phase:** the intermediate key-value pairs are grouped by key and a user-specified reduce function is applied separately to each group of key-value pairs with the same key, producing $\geq 0$ other key-value pairs, which is the output of the round.

- The output of a round is the input of the next round.
MapReduce computation
**Why key-value pairs?**

The use of key-value pairs in MapReduce seems an unnecessary complication, and one might be tempted to regard individual data items simply as *objects* belonging to some domain, without imposing a distinction between keys and values.

**Justification.** MapReduce is data-centric and focuses on data transformations which are independent of where data actually reside. The keys are needed as *addresses* to reach the objects, and as *labels* to define the groups in the reduce phases.

**Practical considerations.**

- One should choose the key-value representations in the most convenient way. The domains for key and values may change from one round to another.
- Programming frameworks such as Spark, as we will see, while containing explicit provisions for handling key-value pairs, allow also to manage data without the use of keys. In this case, internal addresses/labels, not explicitly visible to the programmer, are used.
Implementation of a round

• Input file is split into $X$ chunks and each chunk forms the input of a map task.

• Each map task is assigned to a worker (a compute node) which applies the map function to each key-value pair of the corresponding chunk, buffering the intermediate key-value pairs it produces in its local disk.

• The intermediate key-values pairs, while residing in the local disks, are partitioned into $Y$ buckets through a hash function $h$:

$$ (k, v) \rightarrow \text{Bucket } i = h(k) \mod Y. $$

• Each bucket forms the input of a different reduce task which is assigned to a worker (a compute node).

**Obs.:** hashing helps balancing the load of the workers

The worker applies the reduce function to each group of key-value pairs with the same key, writing the output on the DFS. The application of the reduce function to a group is called reducer.
Implementation of a round

• The user program is forked into a master process and several worker processes. The master is in charge of assigning map and reduce tasks to the various workers, and to monitor their status (idle, in-progress, completed).

• Input and output files reside on a Distributed File System, while intermediate data are stored on the workers’ local disks.

• The round involves a data shuffle for moving the intermediate key-value pairs from the compute nodes where they were produced (by map tasks) to the compute nodes where they must be processed (by reduce tasks).

  **Obs.:** *shuffle is often the most expensive operation of the round*

• The values $X$ and $Y$ are design parameters.
Implementation of a round

From the original paper (split $i \equiv$ chunk $i$):

![Diagram of a round implementation](image-url)
Dealing with faults

- The Distributed File System is fault-tolerant
- Master pings workers periodically to detect failures
- Worker failure:
  - Map tasks completed or in-progress at failed worker are reset to idle and will be rescheduled. Note that even if a map task is completed, the failure of the worker makes its output unavailable to reduce tasks, hence it must be rescheduled.
  - Reduce tasks in-progress at failed worker are reset to idle and will be rescheduled.
- Master failure: the whole MapReduce task is aborted
Specification of a MapReduce algorithm

A *MapReduce (MR) algorithm* should be specified so that

- The input and output of the algorithm is clearly defined
- The sequence of rounds executed for any given input instance is unambiguously implied by the specification
- For each round the following aspects are clear
  - input, intermediate and output sets of key-value pairs
  - functions applied in the map and reduce phases.
- Meaningful values (or asymptotic) bounds for the key performance indicators (defined later) can be derived.
Specification of a MapReduce algorithm

For example, an MR algorithm with a fixed number of rounds $R$, we can use the following style:

**Input:** description of the input as set of key-value pairs

**Output:** description of the output as set of key-value pairs

**Round 1:**

- *Map phase:* description of the function applied to each key-value pair
- *Reduce phase:* description of the function applied to each group of key-value pairs with the same key

Round 2: as before ..

...  

Round R: as before ...

**Observation:** *for simplicity, we sometime provide a high-level description of a round, which, however, must enable a straightforward yet tedious derivation of the Map and Reduce phases.*
Analysis of a MapReduce algorithm

The analysis of an MR algorithm aims at estimating the following key performance indicators:

- **Number of rounds** $R$.
- **Local space** $M_L$: maximum amount of main memory required, in any round, by a single invocation of the map or reduce function used in that round, for storing the input and any data structure needed by the invocation.
- **Aggregate space** $M_A$: maximum amount of (disk) space which is occupied by the stored data at the beginning/end of the map or reduce phase of any round.

**Observations:**

- The indicators are usually estimated through asymptotic analysis (either worst-case or probabilistic) as functions of the instance size.
- $M_L$ bounds the amount of main memory required at each worker, while $M_A$ bounds the overall amount of (disk) space that the executing platform must provide.
Example: Word count

**Input:** collection of text documents $D_1, D_2, \ldots, D_k$ containing $N$ words occurrences (counting repetitions). Each document is a key-value pair, whose key is the document’s name and the value is its content.

**Output:** The set of pairs $(w, c(w))$ where $w$ is a word occurring in the documents, and $c(w)$ is the number of occurrences of $w$ in the documents.

**Round 1:**

- *Map phase:* for each document $D_i$, produce the set of intermediate pairs $(w, c_i(w))$, one for each word $w \in D_i$, where $c_i(w)$ is the number of occurrences of $w$ in $D_i$.

- *Reduce phase:* For each word $w$, gather all intermediate pairs $(w, c_i(w))$ and return the output pair $(w, c(w))$ where $c(w)$ is the sum of the $c_i(w)$’s.

**Observation:** the output, in this case, is the set of pairs $(w, c(w))$ produced by the various reducers, hence one pair for each word.
Analysis of Word count

Let $N_i$ be the number of words in $D_i$ counting repetitions (hence, $N = \sum_{i=1}^{k} N_i$), and assume that each word takes constant space. Let also $N_{\text{max}} = \max_{i=1,k} N_i$. We have:

- $R = 1$ (trivial)
- $M_L = O(\max\{N_{\text{max}}, k\}) = O(N_{\text{max}} + k)$. The bound is justified as follows. In the map phase, the worker processing $D_i$ needs $O(N_i)$ space. In the reduce phase, the worker in charge of word $w$ adds up the contribution of at most $k$ pairs $(w, c_i(w))$.
- $M_A = O(N)$, since the input size is $O(N)$ and $O(N)$ additional pairs are produced by the algorithm, overall.

Observation: in principle, we may have $N_{\text{max}} + k = \Theta(N)$, which implies linear local space. When this happens, most of the advantages of MapReduce vanish (see more in the next two slides). In practice, however it is likely that $k, N_{\text{max}} = o(N)$, so the above algorithm is efficient. We’ll see later how to improve the algorithm ensuring sublinear $M_L$ even for large $k$. 
Design goals for MapReduce algorithms

Here is a simple yet important observation.

**Theorem**

For every computational problem solvable by a sequential algorithm in space $S(|input|)$ there exists a 1-round MapReduce algorithm with $M_L = M_A = \Theta(S(|input|))$

**Proof.**

Run sequential algorithm on whole input with one reducer.

**Remark:** the trivial solution implied by the above theorem is impractical for very large inputs for the following reasons:

- A platform with very large main memory is needed.
- No parallelism is exploited.
Design goals for MapReduce algorithms

In general, to process efficiently very large input instances, an algorithm should aim at: *breaking the computation into a (hopefully small) number of rounds that execute several tasks in parallel, each task working efficiently on a (hopefully small) subset of the data.*

In MapReduce terms, the design goals are the following:

<table>
<thead>
<tr>
<th>Design Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Few rounds (e.g., $R = O(1)$)</td>
</tr>
<tr>
<td>• Sublinear local space (e.g., $M_L = O(</td>
</tr>
<tr>
<td>• Linear aggregate space (i.e., $M_A = O(</td>
</tr>
<tr>
<td>• Polynomial complexity of each map or reduce function</td>
</tr>
</tbody>
</table>

**Observation:** algorithms usually exhibit tradeoffs between performance indicators. Often, small $M_L$ enables high parallelism but may incur large $R$. 
Pros and cons of MapReduce

Why is MapReduce suitable for big data processing?

• **Data-centric view.** Algorithm design can focus on data transformations, targeting the aforementioned design goals. Moreover, programming frameworks supporting MapReduce (e.g., Spark) usually make allocation of tasks to workers, data management, handling of failures, totally transparent to the programmer.

• **Portability/Adaptability.** Applications can run on different platforms and the underlying implementation of the framework will do best effort to exploit parallelism and minimize data movement costs.

• **Cost.** MapReduce applications can be run on moderately expensive platforms, and many popular cloud providers support their execution.
Pros and cons of MapReduce

What are the main drawbacks of MapReduce?

- **Weakness of the number of rounds as performance measure.** It ignores the runtimes of map and reduce functions and the actual volume of data shuffled. More sophisticated (yet, less usable) performance measures exist.

- **Curse of the last reducer.** In some cases, one or a few reducers may be much slower than the other ones, thus delaying the end of the round. When designing MapReduce algorithms one should try to ensure load balancing (if at all possible) among reducers.
Basic Techniques and Primitives
When some aggregation functions (e.g., sum of the $c_i(w)$’s for every word $i$) may potentially receive large inputs (e.g., large $k$) or skewed ones, it is advisable to partition the input, either deterministically or randomly, and perform the aggregation in stages.

We will see two examples:

- An improved version of Word count
- A Class count primitive
Improved Word count

Consider the word count problem, $k$ documents and $N$ word occurrences overall, in a realistic scenario where $k = \Theta(N)$ and $N$ is huge (e.g., huge number of small documents). The following algorithm reduces local space requirements at the expense of an extra round.

**Idea:** partition intermediate pairs randomly in $o(N)$ groups and compute counts in two stages

**Round 1:**

- **Map phase:** for each document $D_i$, produce the intermediate pairs $(x, (w, c_i(w)))$, one for every word $w \in D_i$, where $x$ (the key of the pair) is a random integer in $[0, \sqrt{N}]$ and $c_i(w)$ is the number of occurrences of $w$ in $D_i$.

- **Reduce phase:** For each key $x$ gather all pairs $(x, (w, c_i(w)))$, and for each word $w$ occurring in these pairs produce the pair $(w, c(x, w))$ where $c(x, w) = \sum_{(x, (w, c_i(w)))} c_i(w)$. Now, $w$ is the key for $(w, c(x, w))$. 
Improved Word count

Round 2:

- **Map phase**: empty.
- **Reduce phase**: for each word $w$, gather the at most $\sqrt{N}$ pairs $(w, c(x, w))$ resulting at the end of the previous round, and return the output pair $(w, \sum_x c(x, w))$.

**Analysis.** Let $m_x$ be the number of intermediate pairs with key $x$ produced by the Map phase of Round 1, and let $m = \max_x m_x$. As before, let $N_i$ be the number of words in $D_i$. We have

- $R = 2$
- $M_L = O \left( \max_{i=1,k} N_i + m + \sqrt{N} \right)$.
- $M_A = O(N)$

**How large can $m$ be?** We need a very useful technical tool.
Chernoff bound

Let $X_1, X_2, \ldots, X_n$ be $n$ i.i.d. Bernoulli random variables, with $\Pr(X_i = 1) = p$, for each $1 \leq i \leq n$. Thus, $X = \sum_{i=1}^{n} X_i$ is a Binomial$(n, p)$ random variable. Let $\mu = E[X] = n \cdot p$. For every $\delta_1 \geq 5$ and $\delta_2 \in (0, 1)$ we have that

$$\Pr(X \geq (1 + \delta_1)\mu) \leq 2^{-(1+\delta_1)\mu}$$
$$\Pr(X \leq (1 - \delta_2)\mu) \leq 2^{-\mu\delta_2^2/2}$$

The proof can be found in [MU05].
# Bound on $m$ for Improved Word count

## Theorem

Suppose that the keys assigned to the intermediate pairs in Round 1 are i.i.d. random variables with uniform distribution in $[0, \sqrt{N})$. Then, with probability at least $1 - \frac{1}{N^5}$

$$m = O\left(\sqrt{N}\right).$$

Therefore, from the theorem and the preceding analysis we get

$$M_L = O\left(\max_{i=1,k} N_i + \sqrt{N}\right),$$

with probability at least $1 - \frac{1}{N^5}$. In fact, for large $N$ the probability becomes very close to 1.

**N.B.:** This is an example of probabilistic analysis, as opposed to more traditional worst-case analysis.
Bound on $m$ for Improved Word count

Proof of theorem.

Let $N' \leq N$ be the number of intermediate pairs produced by the Map phase of Round 1, and consider an arbitrary key $x \in [0, \sqrt{N})$.

**Crucial observation:** $m_x$ is a Binomial($N', 1/\sqrt{N}$) random variable with expectation $\mu = N'/\sqrt{N} \leq \sqrt{N}$. Since $6\sqrt{N} \geq 6\mu$, by the Chernoff bound

$$\Pr(m_x \geq 6\sqrt{N}) \leq \frac{1}{2^{6\sqrt{N}}} \leq \frac{1}{2^{6\log_2 N}} = \frac{1}{N^6}.$$ 

Now, the event $"m \geq 6\sqrt{N}"$ is the union of the events $"m_x \geq 6\sqrt{N}"$, over all $x$. By union bound (the probability of a union of events is upper bounded by the sum of the probabilities of the individual events) we have that

$$\Pr(m \geq 6\sqrt{N}) \leq \sum_{x \in [0, \sqrt{N})} \Pr(m_x \geq 6\sqrt{N}) \leq \sqrt{N} \Pr(m_x \geq 6\sqrt{N}) \leq \frac{1}{N^5}.$$ 

Therefore, with probability at least $1 - 1/N^5$ we have $m \leq 6\sqrt{N}$, i.e.,

$m = O(\sqrt{N})$. \qed
Observations

The choice of partitioning the intermediate pairs into groups of size $O\left(\sqrt{N}\right)$, in the first round of Improved Word count, is somewhat arbitrary, but it provides an example of a simple tradeoff between local space and number of rounds.

In the rest of the course we will often make a similar choice. However, it is important to understand that the approach can be generalized to attain a given target bound on $M_L$ (e.g., $M_L = O\left(N^\epsilon\right)$ for some $\epsilon \in (0, 1)$) exercising a finer tradeoff between $M_L$ and number of rounds. One of the exercises will explore this issue.
Class count

Suppose that we are given a set $S$ of $N$ objects, each labeled with a class from a given domain, and we want to count how many objects belong to each class.

More precisely:

**Input:** Set $S$ of $N$ objects represented by pairs $(i, (o_i, \gamma_i))$, for $0 \leq i < N$, where $o_i$ is the $i$-th object, and $\gamma_i$ its class.

**Output:** The set of pairs $(\gamma, c(\gamma))$ where $\gamma$ is a class labeling some object of $S$ and $c(\gamma)$ is the number of objects of $S$ labeled with $\gamma$.

**Observation:** the straightforward 1-round algorithm may require $O(N)$ local space, in case a large fraction of objects belong to the same class. In the next slide we will see a more efficient algorithm.
Class count

Round 1:

• **Map phase:** map each pair \((i, (o_i, \gamma_i))\) into the intermediate pair 
  \((i \mod \sqrt{N}, (o_i, \gamma_i))\) (mod = reminder of integer division)

• **Reduce phase:** For each key \(j \in [0, \sqrt{N}]\) gather the set (say \(S^{(j)}\)) of all intermediate pairs with key \(j\) and, for each class \(\gamma\) labeling some object in \(S^{(j)}\), produce the pair \((\gamma, c_j(\gamma))\), where \(c_j(\gamma)\) is the number of objects of \(S^{(j)}\) labeled with \(\gamma\).

Round 2:

• **Map phase:** empty

• **Reduce phase:** for each class \(\gamma\), gather the at most \(\sqrt{N}\) pairs
  \((\gamma, c_j(\gamma))\) resulting at the end of the previous round, and return the output pair \((\gamma, \sum_j c_j(\gamma))\).

**Exercise**

Derive bounds on \(M_L\) and \(M_A\) for the above algorithm.
Trading accuracy for efficiency

There are problems for which exact MR algorithms may be too costly, namely they may require a large number of rounds, or large (i.e., close to linear) local space, or large (i.e., superlinear) aggregate space. These algorithms become impractical for very large inputs.

In these scenarios, giving up exact solutions (if acceptable for the application) may greatly improve efficiency.

We’ll now see an example through an important primitive for the processing of pointsets from metric spaces (e.g., spatial datasets).
Maximum pairwise distance

Suppose that we are given a set $S$ of $N$ points from some metric space (e.g., $\mathbb{R}^3$) and we want to determine the maximum distance between two points, for a given distance function $d(\cdot, \cdot)$.

More precisely:

**Input:** Set $S$ of $N$ points represented by pairs $(i, x_i)$, for $0 \leq i < N$, where $x_i$ is the $i$-th point.

**Output:** $(0, \max_{0 \leq i, j < N} d(x_i, x_j))$.

**Exercise**

Design an MR algorithm for the problem which requires

$R = O(1)$, $M_L = O\left(\sqrt{N}\right)$ and $M_A = O\left(N^2\right)$. 
Maximum pairwise distance

Pointset S

Maximum pairwise distance
Maximum pairwise distance

We can substantially reduce the aggregate space requirements if we tolerate a factor 2 error in the estimate of $d_{\text{max}}$. For an arbitrary point $x_i$ define

$$d_{\text{max}}(i) = \max_{0 \leq j < N} d(x_i, x_j).$$

**Lemma**

For any $0 \leq i < N$ we have $d_{\text{max}} \in [d_{\text{max}}(i), 2d_{\text{max}}(i)]$.

**Proof.**

It is immediate to see that $d_{\text{max}} \geq d_{\text{max}}(i)$. Suppose that $d_{\text{max}} = d(x_r, x_t)$ for some $r, t$. By the triangle inequality we have

$$d(x_r, x_t) \leq d(x_r, x_i) + d(x_i, x_t) \leq 2d_{\text{max}}(i).$$
Maximum pairwise distance

Round 1:

- **Map phase:** map each pair \((i, x_i)\) into the intermediate pair \((i \mod \sqrt{N}, (i, x_i))\). Also, create the \(\sqrt{N} - 1\) pairs \((j, (0, x_0))\), for \(1 \leq j < \sqrt{N}\).

- **Reduce phase:** For each key \(j \in [0, \sqrt{N})\) gather the set \(S^{(j)}\) of all intermediate pairs with key \(j\) (which include \((j, (0, x_0))\)) and produce the pair \((0, d_{\text{max}}(0, j))\) where \(d_{\text{max}}(0, j)\) is the maximum distance between \(x_0\) and the points associated with pairs in \(S^{(j)}\).

Round 2:

- **Map phase:** empty.

- **Reduce phase:** gather all pairs \((0, d_{\text{max}}(0, j))\) with \(j \in [0, \sqrt{N})\) and return the output pair \((0, \max_{0 \leq j < \sqrt{N}} d_{\text{max}}(0, j))\).

**Analysis:** \(R = 2\), \(M_L = O\left(\sqrt{N}\right)\), \(M_A = O(N)\).
Exploiting samples

When facing a big-data processing task, we should ask ourselves the following question:

Can we profitably process a small sample of the data?

In many cases, the answer is YES! Specifically, a small sample, suitably extracted, could be exploited for the following purposes.

- To subdivide the dataset in smaller subsets to be analyzed separately.
- To provide a succinct yet accurate representation of the whole dataset, which contains a good solution to the problem and filters out noise and outliers, thus allowing the execution of the task on the sample.

In what follows, we will see an example of the first type through the sorting primitive, while an example of the second type will be given later.
Sorting

Input: Set $S = \{s_i : 1 \leq i \leq N\}$ of $N$ distinct sortable objects (each $s_i$ represented as a pair $(i, s_i)$)

Output: Sorted set $\{(i, s_{\pi(i)}) : 1 \leq i \leq N\}$, where $\pi$ is a permutation such that $s_{\pi(1)} \leq s_{\pi(2)} \leq \cdots \leq s_{\pi(N)}$.

The MR SampleSort algorithm is based on the following idea:

- Fix a suitable integral design parameter $K$.
- Randomly select some input objects ($K$ on average) as splitters.
- Partition the objects into subsets based on the ordered sequence of splitters.
- Sort each subset separately and compute the final ranks.

Obs.: SampleSort is at the base of TeraSort, a popular benchmark for measuring the performance of computing platforms, as well as most winners of the big-data Sort Benchmark competition.
MR SampleSort

Round 1:

- **Map phase**: for each pair \((i, s_i)\) do the following:
  
  - transform the pair into \((i \mod K, (0, s_i))\) (call it *regular pair*);
  - with probability \(p = K/N\), independently of other objects, select \(s_i\) as a *splitter* and, if selected, create \(K\) *splitter pairs* \((j, (1, s_i))\), with \(0 \leq j < K\). (Note that a binary flag is used to distinguish between splitter from regular pairs).

**Obs.** Let \(t\) be the number of splitters selected in the Map phase. At the end of the phase, the splitter pairs represent \(K\) copies of the splitters, one for each key \(j\).

- **Reduce phase**: for \(0 \leq j < K\) do the following: gather all regular and splitter pairs with key \(j\); sort the \(t\) splitters and let \(x_1 \leq x_2 \leq \cdots \leq x_t\) be the splitters in sorted order. Transform each regular pair \((j, (0, s))\) into the pair \((\ell, s)\), where \(\ell \in [0, t]\) is such that \(x_\ell < s \leq x_{\ell+1}\) (assume \(x_0 = -\infty\) and \(x_{t+1} = +\infty\)).
MR SampleSort

- **Round 2:**
  - *Map phase:* empty
  - *Reduce phase:* for every $0 \leq \ell \leq t$ gather, from the output of the previous round, the set of pairs $S^{(\ell)} = \{(\ell, s)\}$, compute $N_\ell = |S^{(\ell)}|$, and create $t + 1$ replicas of $N_\ell$ (use suitable pairs for these replicas).

- **Round 3:**
  - *Map phase:* empty
  - *Reduce phase:* for every $0 \leq \ell \leq t$ do the following: gather $S^{(\ell)}$ and the values $N_0, N_1, \ldots, N_t$; sort $S^{(\ell)}$; and compute the final output pairs for the objects in $S^{(\ell)}$ whose ranks start from $1 + \sum_{h=0}^{\ell-1} N_h$. 
Example

\[ N = 32, \ K = 4 \]

\[ S = 16, 12, 25, 31, 32, 3, 21, 22, 1, 29, 13, 9, 15, 17, 5, 6, \]
\[ 14, 11, 19, 27, 7, 10, 23, 24, 28, 8, 30, 4, 20, 2, 26, 18 \]

Round 1. Call \( S_j \) the set of intermediate pairs \( (j, (0, s)) \) after the Map phase. We have (objects only):

\[ S_0 = 16, 32, 1, 15, 14, 7, 28, 20 \]
\[ S_1 = 12, 3, 29, 17, 11, 10, 8, 2 \]
\[ S_2 = 25, 21, 13, 5, 19, 23, 30, 26 \]
\[ S_3 = 31, 22, 9, 6, 27, 24, 4, 18 \]

The \( t = 5 \) splitters are highlighted in blue. In sorted order:

\[ x_1 = 4, x_2 = 9, x_3 = 16, x_4 = 21, x_5 = 29. \]
Example

Round 1 (cont’d). Call $S_j^{(ℓ)}$ the set of intermediate pairs $(j, (0, s))$ with $x_ℓ < s ≤ x_{ℓ+1}$. We have (objects only):

<table>
<thead>
<tr>
<th>$j$</th>
<th>$S_j^{(0)}$</th>
<th>$S_j^{(1)}$</th>
<th>$S_j^{(2)}$</th>
<th>$S_j^{(3)}$</th>
<th>$S_j^{(4)}$</th>
<th>$S_j^{(5)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>16,15,14</td>
<td>20</td>
<td>28</td>
<td>32</td>
</tr>
<tr>
<td>1</td>
<td>3,2</td>
<td>8</td>
<td>12,11,10</td>
<td>17</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>9,6</td>
<td>13</td>
<td>21,19</td>
<td>25,23,26</td>
<td>31</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td>18</td>
<td>22,27,24</td>
<td></td>
</tr>
</tbody>
</table>

Round 2 The $S^{(ℓ)}$’s (objects only) and the $N_ℓ$’s values are

$S^{(0)} = 1, 3, 2, 4$  $N_0 = 4$
$S^{(1)} = 7, 8, 5, 9, 6$  $N_1 = 5$
$S^{(2)} = 16, 15, 14, 12, 11, 10, 13$  $N_2 = 7$
$S^{(3)} = 20, 17, 21, 19, 18$  $N_3 = 5$
$S^{(4)} = 28, 29, 25, 23, 26, 22, 27, 24$  $N_4 = 8$
$S^{(5)} = 32, 30, 31$  $N_5 = 3$
Example

Round 3. Final output

(1, 1), \cdots, (4, 4) \quad \text{from rank 1}
(5, 5), \cdots, (9, 9) \quad \text{from rank } N_0 + 1 = 5
(10, 10), \cdots, (16, 16) \quad \text{from rank } N_0 + N_1 + 1 = 10
(17, 17), \cdots, (21, 21) \quad \text{from rank } N_0 + N_1 + N_2 + 1 = 17
(22, 22), \cdots, (29, 29) \quad \text{from rank } N_0 + N_1 + N_2 + N_3 + 1 = 22
(30, 30), \cdots, (32, 32) \quad \text{from rank } N_0 + \cdots + N_4 + 1 = 30
Analysis of MR SampleSort

- Number of rounds: \( R = 3 \)
- Local Space \( M_L \):
  - Round 1: \( O(K + t + N/K) \), since \( K \) copies of each splitter are created, and each reducer must store all splitter pairs and a subset of \( N/K \) intermediate pairs.
  - Round 2: \( O(t + 1 + \max\{N_\ell ; 0 \leq \ell \leq t\}) \) since each reducer must gather one \( S^{(\ell)} \) and make \( t + 1 \) replicas of its size.
  - Round 3: \( O(t + \max\{N_\ell ; 0 \leq \ell \leq t\}) \), since each reducer must store all \( N_\ell \)'s and one \( S^{(\ell)} \).

\[ \Rightarrow \text{overall } M_L = O(K + t + N/K + \max\{N_\ell ; 0 \leq \ell \leq t\}) \]

- Aggregate Space \( M_A \): \( O(N + t \cdot K + t^2) \), since in Round 1 each splitter is replicated \( K \) times, and in Round 3 each \( N_\ell \) is replicated \( t + 1 \) times. The objects are never replicated.
Analysis of MR SampleSort

Lemma

With reference to the MR SampleSort algorithm, for any $K \in (2\ln N, N)$ the following two inequalities hold with high probability (i.e., probability at least $1 - 1/N$):

1. $t \leq 6K$, and
2. $\max\{N_\ell ; 0 \leq \ell \leq t\} \leq 4(N/K) \ln N$.

Proof.

Deferred.

Theorem

By setting $K = \sqrt{N}$, MR SampleSort runs in 3 rounds, and, with high probability, it requires local space $M_L = O(\sqrt{N} \ln N)$ and aggregate space $M_A = O(N)$.

Proof.

Immediate from the lemma and the previous analysis.
Analysis of MR SampleSort

Proof of Lemma

We show that each inequality holds with probability at least $1 - 1/(2N)$.

1. $t$ is a Binomial($N, K/N$) random variable with $E[t] = K > 2\ln N$. By the Chernoff bound (first inequality with $\delta_1 = 5$) we have that $t > 6K$ with probability at most $1/(2N)$.

2. View the sorted sequence of objects as divided into $K/(2\ln N)$ contiguous segments of length $N' = 2(N/K)\ln N$ each, and consider one arbitrary such segment. The probability that no splitter is chosen among the objects of the segment is

\[
\left(1 - \frac{K}{N}\right)^{N'} = \left(1 - \frac{1}{(N/K)}\right)^{(N/K)2\ln N} \leq \left(\frac{1}{e^{\ln N}}\right)^2 = \frac{1}{N^2}.
\]

(We used the well-known inequality $(1 - 1/x)^x \leq 1/e$, $\forall x \geq 0.$)
Analysis of MR SampleSort

Sorted sequence of N objects

Segment of $2(N/K) \ln N$ objects

$\leq 4(N/K) \ln N$ objects

Splitter
Proof of Lemma.

(continuation) So, there are $K/(2 \ln N)$ segments and we know that, for each segment, the event “no splitter falls in the segment” occurs with probability $\leq 1/N^2$. Hence, the probability that any of these $K/(2 \ln N)$ events occurs is $\leq K/(N^2 2 \ln N) = 1/(2N)$ (union bound!). Therefore, with probability at least $(1 - 1/(2N))$, at least 1 splitter falls in each segment, which implies that each $N_\ell$ cannot be larger than $4(N/K) \ln N$. Hence, we have that the second inequality stated in the lemma holds with probability at least $(1 - 1/(2N))$.

In conclusion, we have that the probability that at least one of the two inequalities does not hold, is at most $2 \cdot 1/(2N) = 1/N$, and the lemma follows.
Exercises

Exercise 1

For each of the 4 problems listed in Exercise 2.3.1 of [LRU14], design an $O(1)$-round MR algorithm which uses $O(\sqrt{N})$ local space and linear aggregate space, where $N$ is the input size.

Exercise 2

Suppose that an instance of the Class count problem is given consisting only of object-class pairs $(o, \gamma)$ (i.e., without the initial integer keys in $[0, N]$). Consider a modification of the algorithm described in the slides, where in the Map phase of Round 1 each pair $(o, \gamma)$ is mapped into a pair $(j, (o, \gamma))$, where $j$ is a random integer in $[0, \sqrt{N}]$. The rest of the algorithm is unchanged. Analyze probabilistically the local space requirements of the modified algorithm.
Exercises

Exercise 3
Design and analyze an efficient MR algorithm for approximating the maximum pairwise distance, which uses local space $M_L = O(N^{1/4})$. Can you generalize the algorithm to attain $M_L = O(N^\epsilon)$, for any $\epsilon \in (0, 1/2)$?

Exercise 4
Design an $O(1)$-round MR algorithm for computing a matrix-vector product $W = A \cdot V$, where $A$ is an $m \times n$ matrix and $V$ is an $n$-vector. Your algorithm must use $o(n)$ local space and linear (i.e., $O(mn)$) aggregate space. For simplicity you may assume $m \leq \sqrt{n}$, but think about how your solution changes if $m$ is larger.
Summary

- MapReduce.
  - Motivating scenario.
  - Main Features, platforms and software frameworks.
  - MapReduce computation: high-level structure, implementation, algorithm specification, analysis, key performance indicators.
  - Algorithm design goals

- Basic Techniques and Primitives.
  - Chernoff bound
  - Partitioning: Word count and Class count.
  - Efficiency-accuracy tradeoffs: Maximum pairwise distance
  - Sampling: Sorting
References


DG08 J. Dean and A. Ghemawat. MapReduce: simplified data processing on large clusters. OSDI’04 and CACM 51,1:107113, 2008


Changes w.r.t. first draft:

- Slide 33: added some passages and clarifications.
- Slide 43: indices of the input/output objects start now from 1; also clarified that splitters are selected from the input objects.
- Slide 46: changed the order of the input points to make it consistent with the groups after map phase.
- Slide 49: added a term $t + 1$ to the contribution of R2 to $M_L$, and $K$ to the final $M_L$ bound.
- Slide 55: added a simplifying assumption to Ex.4.