DSC-204A: Scalable Data Systems, Winter 2024

Streaming Processing-3

Lecturer: Hao Zhang *Scribe: Yunzhou Yan*

Spark and RDD

RDD, or Resilient Distributed Dataset, is a fundamental abstraction in Apache Spark, a powerful distributed computing framework for processing large datasets. RDDs serve as the primary data structure in Spark and provide a distributed, fault-tolerant, and parallelized way to manipulate data across a cluster of machines.

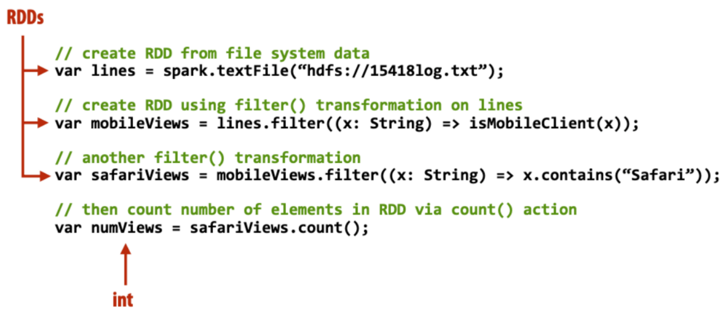
The key characteristics of RDD

Read-only collection of records (immutable)

RDDs are immutable, meaning once created, they cannot be changed. Instead, any transformations or operations applied to an RDD will produce a new RDD. This immutability ensures that RDDs are resilient to errors and can be efficiently distributed across a cluster without worrying about data corruption.

RDDs can only be created by deterministic transformations on data in persistent storage or on existing RDDs

RDDs are created through deterministic transformations on existing data sources or other RDDs. These transformations include operations like map, filter, reduce, join, etc.Deterministic transformations ensure that the same input data will always produce the same output RDD, regardless of the execution environment. This property is crucial for achieving fault tolerance and reproducibility in distributed computing.
Below is an example of how we create and transform RDDs.



```
// create RDD from file system data
var lines = spark.textFile("hdfs://15418log.txt");

// create RDD using filter() transformation on lines
var mobileViews = lines.filter((x: String) => isMobileClient(x));

// another filter() transformation
var safariViews = mobileViews.filter((x: String) => x.contains("Safari"));

// then count number of elements in RDD via count() action
var numViews = safariViews.count();
```

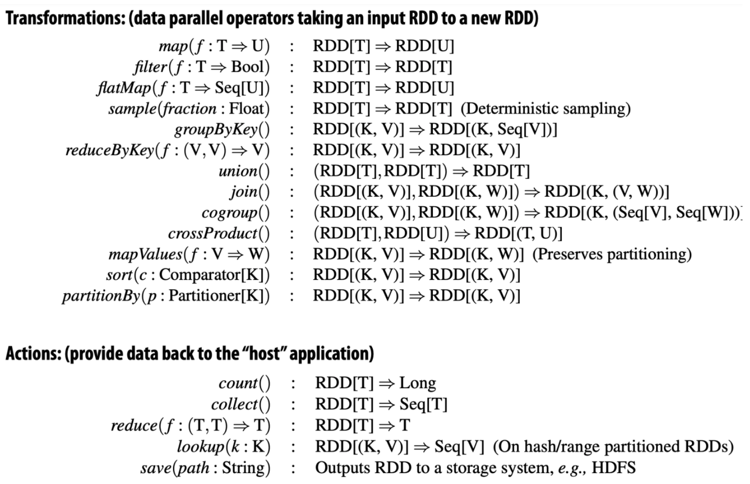
Figure 1: Create and transform RDDs

Transformation and Action

In the RDD programming model, transformations and actions are two fundamental types of operations used to manipulate data. Transformations in Spark are operations applied to an RDD to create a new RDD. These operations are lazy, meaning they do not compute a result immediately but rather build up a logical execution plan that will be executed only when an action is called. Transformations are typically used to perform data processing tasks such as filtering, mapping, aggregating, joining, and sorting. Examples of transformation operations include map, filter, flatMap, reduceByKey, join, sortBy, groupBy, etc. When a transformation is applied to an RDD, Spark optimizes the execution plan and builds a Directed Acyclic Graph representing the sequence of transformations to be executed on the dataset.

Actions in Spark are operations that trigger the execution of the transformation operations and produce a result that is returned to the driver program or written to an external storage system. Unlike transformations, actions are eager operations, meaning they initiate the execution of the previously defined lazy transformations and compute a result. Examples of action operations include collect, count, reduce, take, foreach, saveAsTextFile, saveAsParquetFile, etc. When an action is called on an RDD, Spark evaluates the DAG of transformations and executes them in parallel across the cluster to produce the desired result.

In summary, transformations are used to define the sequence of operations to be applied to the dataset, while actions are used to trigger the execution of these operations and produce results. Many common transformations and actions are shown in the figure below.



Transformations: (data parallel operators taking an input RDD to a new RDD)

map(f : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
filter(f : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
flatMap(f : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
reduceByKey(f : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
mapValues(f : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
sort(c : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
partitionBy(p : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]

Actions: (provide data back to the "host" application)

count()	:	RDD[T] ⇒ Long
collect()	:	RDD[T] ⇒ Seq[T]
reduce(f : (T, T) ⇒ T)	:	RDD[T] ⇒ T
lookup(k : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
save(*path* : String)	:	Outputs RDD to a storage system, *e.g.,* HDFS

Figure 2: Transformations and actions

Spark vs Mapreduce

Flexibility and Expressiveness

Spark offers greater flexibility and expressiveness compared to MapReduce, as it eliminates the necessity of writing programs with explicit map and reduce functions. Many programs have been proven to be impossible to write in terms of map and reduce functions. However, Spark's flexibility and expressiveness are not entirely unrestricted, as users are still required to utilize predefined transformations and actions.

Simplicity

Spark is also simpler than MapReduce, as it eliminates the need for writing programs with explicit map and reduce functions. Moreover, Spark offers a more convenient high-level API. Additionally, Spark's capability to cache intermediate results in memory reduces the requirement for redundant computations, resulting in simpler and more efficient workflows.

Scalability

Spark is designed for in-memory, which allows it to handle larger datasets more efficiently than MapReduce in many cases. Spark's DAG-based execution engine optimizes task scheduling and resource utilization, leading to better scalability for iterative and interactive workloads. MapReduce's reliance on disk I/O for intermediate data storage can lead to performance bottlenecks, especially for iterative algorithms.

Fault tolerance

The fault tolerance of Spark is worse than MapReduce because MapReduce can replicate intermediate data across multiple nodes on disks and re-executing failed tasks. On the other hand, the size of memory is limited.

The implementation of RDD

Storage of RDD

In Spark, RDDs (Resilient Distributed Datasets) are stored and managed across the nodes of a cluster. There are primarily two ways in which RDDs can be stored:

In-memory storage

By default, Spark RDDs are stored in memory whenever possible. This means that once an RDD is computed, Spark keeps the data in memory on the nodes of the cluster. Storing RDDs in memory allows for much faster access to the data compared to reading it from disk every time it's needed.

Disk storage

If the memory on the nodes is not sufficient to store the entire RDD, or if there is a need to spill data to disk due to memory constraints, Spark can also store RDDs on disk. This typically happens when the RDD is too large to fit entirely in memory.

Data structure for RDD

In Apache Spark, RDDs are represented internally using a combination of data structures optimized for distributed computing. The principal data structures employed for storing RDDs are as follows:

Partitioned Collections

RDDs are typically partitioned collections of records distributed across the nodes of a Spark cluster. Each partition represents a subset of the data and is processed independently in parallel. RDDs are typically partitioned collections of records distributed across the nodes of a Spark cluster. Each partition represents a subset of the data and is processed independently in parallel.

RDDs define dependencies between transformations as either narrow or wide. These dependencies play a crucial role in optimizing the execution of Spark jobs across a distributed cluster. Narrow dependencies occur when each partition of the parent RDD is used by at most one partition of the child RDD in a transformation. it allows for the fusing of operations. Narrow dependencies enable Spark to perform transformations locally on each partition without shuffling or redistributing data across the cluster. This optimization reduces the amount of data movement and improves the efficiency of computation. Examples of transformations that typically result in narrow dependencies include map, filter, flatMap, mapPartitions, union, etc.

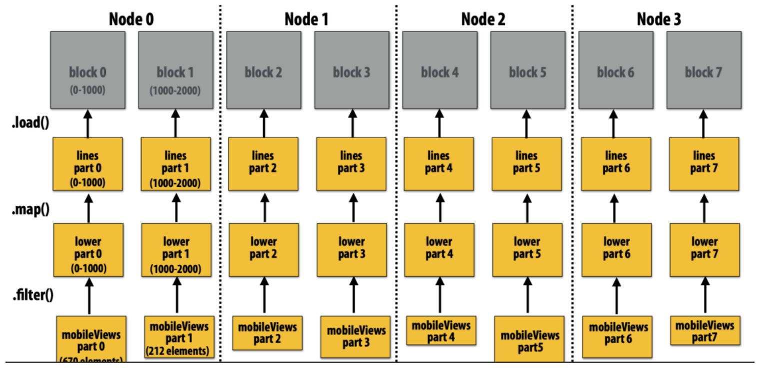


Figure 3: Narrow dependencies

Wide dependencies occur when each partition of the parent RDD may be used by multiple partitions of the child RDD in a transformation. Wide dependencies typically involve operations that require data to be grouped, aggregated, or sorted across partitions, resulting in data shuffling across the cluster. This can incur significant overhead in terms of network communication and disk I/O. For example, groupByKey() may induce all-to-all communication. Wide dependencies may trigger significant recompilation of ancestor lineage upon node failure. Examples of transformations that typically result in wide dependencies include groupByKey, reduceByKey, join, sortByKey, cogroup, etc.

Scheduling Spark computations

In Spark, scheduling computations efficiently is critical for optimizing performance and resource utilization in distributed data processing. When actions like save() are triggered, Spark evaluates the lineage graph of

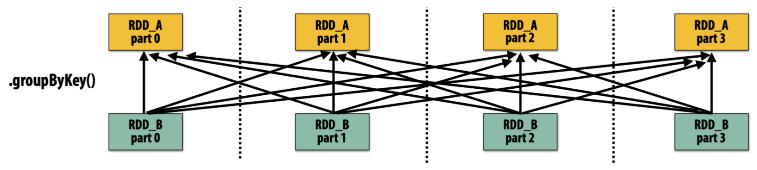


Figure 4: Wide dependencies

RDD transformations to determine the sequence of operations needed to compute the final result.

Stage 1 Computation

In this stage, the input data is already materialized in memory, meaning it's readily available for processing without any additional computation or data movement.

Stage 2 Computation

In this stage, Spark evaluates the map transformation in a fused manner. Fused transformations allow Spark to optimize the execution plan by combining multiple sequential transformations into a single stage, reducing the overhead of passing data between stages. Only the final RDD, let's call it RDD F, is actually materialized as a result of the fused map transformation. This optimization helps minimize the amount of intermediate data materialized in memory, thereby reducing memory usage and improving performance.

Stage 3 Computation

In this stage, Spark executes the join operation. However, instead of materializing the entire result of the join operation in memory, Spark may choose to stream the operation to disk if necessary. This means that Spark can perform the join operation without fully materializing the intermediate result, thereby reducing memory pressure and improving scalability. By streaming the join operation to disk, Spark can handle larger datasets that may not fit entirely in memory, while still maintaining efficient computation and fault tolerance.

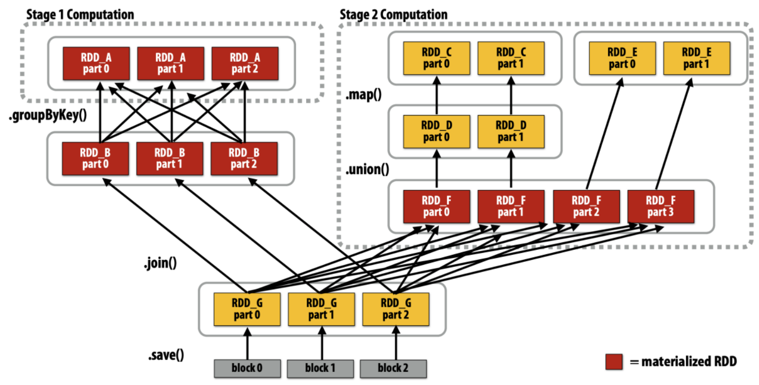


Figure 5: The evaluation of Spark lineage graph for save action

Implementing resilience via lineage

RDD transformations in Spark are bulk, deterministic, and functional.The key implication of these characteristics is that Spark can reconstruct the contents of an RDD from its lineage. which serves as a log of transformations, capturing the steps needed to derive a particular RDD. Lineage is an efficient mechanism for achieving fault tolerance in Spark. Since lineage records bulk data-parallel operations rather than fine-grained operations, the overhead of logging is relatively low compared to systems like databases. By storing the lineage information, Spark can recover lost RDD partitions by re-executing the sequence of transformations that led to their creation. This process involves reloading the required subset of data from disk and recomputing the operations specified in the lineage.

Pros and Cons of Spark

Pros

- Easy for programmers because computation is expressed by chaining atomic operators.
- Much fewer I/O, leading to significantly improved performance, especially in AI applications.

Cons

- Debuggability may be challenging.
- Spark can be perceived as bulky.
- MapReduce may be preferable in single-worker scenarios, leading to the ubiquity of "map" functions in programming languages.

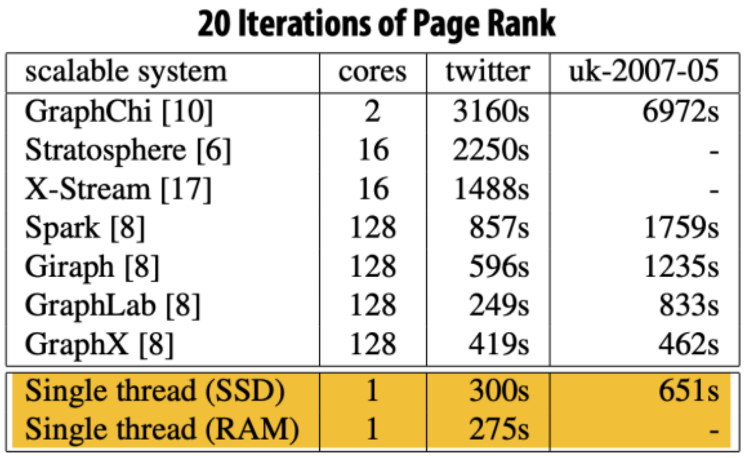
Caution: "Scale out" is not the entire story. Distributed systems designed for cloud execution address many difficult challenges and have been instrumental in the explosion of big-data computing and large-scale analytics. They include:

- Scale-out parallelism to many machines.
- Resiliency in the face of failures.
- Complexity of managing clusters of machines.

But scale out is not the whole story: Sometimes we want our computer to execute simple tasks (e.g. PageRank), where Spark can perform much worse than single thread workers because of the time costs in managing RDD.

Story time: Spark and Databricks

Once upon a time, Spark emerged as an open-source project initiated by a small group of students. As its capabilities outshone those of its predecessors, Hadoop and MapReduce, its community flourished. However,



20 Iterations of Page Rank

scalable system	cores	twitter	uk-2007-05
GraphChi [10]	2	3160s	6972s
Stratosphere [6]	16	2250s	-
X-Stream [17]	16	1488s	-
Spark [8]	128	857s	1759s
Giraph [8]	128	596s	1235s
GraphLab [8]	128	249s	833s
GraphX [8]	128	419s	462s
Single thread (SSD)	1	300s	651s
Single thread (RAM)	1	275s	-

Figure 6: For PageRank tasks, Spark performs worse than single-thread workers

as these student developers were on the cusp of graduation, they faced a dilemma: they couldn't commit to sustaining the project.

Seeking a solution, they approached Hortonworks, proposing that the company take over the project. However, Hortonworks declined the opportunity. Undeterred, the students embarked on a new journey and founded Databricks.

Meanwhile, Cloudera, a data platform company founded by Hadoop authors, once stood tall as a unicorn in the tech industry. Yet, it faced formidable challenges from rising stars like Databricks and Snowflake. Despite going public in 2017, Cloudera's stock price dwindled, leading to a merger with Hortonworks in 2018. Eventually, in 2021, Cloudera transitioned to private ownership after being acquired by investment firms.

Databricks, led by seven co-founders with Prof. Ion Stoica at the helm initially struggled to find its footing. Despite attempts to sell Spark, their efforts were fruitless. It wasn't until Ali Ghodsi, an Iranian-Swedish entrepreneur, took the reins that Databricks began to see progress. Facing numerous challenges, including a near-collapse during 2018-2020, Databricks persisted and raised funds through Series I financing rounds.

However, Databricks found its breakthrough in data warehousing and OLAP (Online Analytical Processing) services. With competitors faltering, Databricks seized the opportunity to educate customers and capitalize on the ever-expanding volumes of data.

Despite setbacks, Databricks aimed to go public in 2022, only to have their plans disrupted by the COVID-19 pandemic. Nevertheless, as of today, Databricks commands a valuation of 43 billion, raising questions about market bubbles. The company's success has even birthed three billionaires. Amidst fierce competition with Snowflake, Databricks continues to thrive in the ever-evolving landscape of data analytics and processing.

Machine learning system

The difficulty of ML system research

Machine learning algorithms are very diverse. ML was still a mess in 2012 when we have XGBOOST, Spark mLib, LDA and Theano

Diversity: Good News or Bad News?

Machine Learning (ML) is incredibly diverse, offering both challenges and opportunities.

Cons:

- There is no unified model or computation, making it challenging to standardize approaches across different applications.
- It's difficult to build a programming model or interface that covers the diverse range of ML applications due to their varying requirements and characteristics.
- Identifying system bottlenecks is challenging due to the complexity and diversity of ML algorithms and workflows.

Pros:

- Despite the challenges, the diversity in ML presents a multitude of opportunities, akin to a gold mining era, where exploration and discovery abound.

In summary, diversity in machine learning offers both challenges and opportunities, requiring innovative approaches and solutions to navigate the complex landscape effectively.

First unification: Gradient Descent

Most ML algorithms (including the popular neural network) are iterative-convergent. Gradient descent is the master equation behind it. The common gradient descent update can be expressed as: The gradient

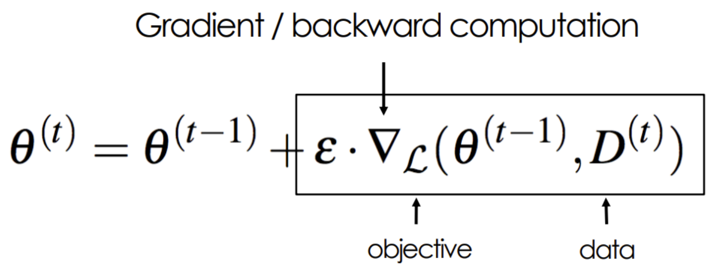

Gradient / backward computation
$\theta^{(t)} = \theta^{(t-1)} + \boxed{\varepsilon \cdot \nabla_{\mathcal{L}}(\theta^{(t-1)}, D^{(t)})}$
objective
data

Figure 7: Gradient Descent Update

descent update is equivalent to the update formula below. We can see that the computation of gradients can be distributed to many different workers.
Using Spark to address this issue presents the following challenges:

- Very heavy communication per iteration because of the large number of parameters, which can significantly impact performance and scalability.
- The compute-to-communication ratio is approximately 1:10 in the era of 2012, indicating a potential imbalance in resource utilization.

$$\theta^{(t+1)} = \theta^{(t)} + \varepsilon \sum_{p=1}^{P} \nabla_{\mathcal{L}}(\theta^{(t)}, D_p^{(t)})$$

Figure 8: Gradient Descent Update

Additionally, consistency in parameters is crucial. Ensuring that parameters across different workers remain consistent is vital to guaranteeing uniformity before updates are applied at each iteration. Otherwise, machine learning models may not converge.

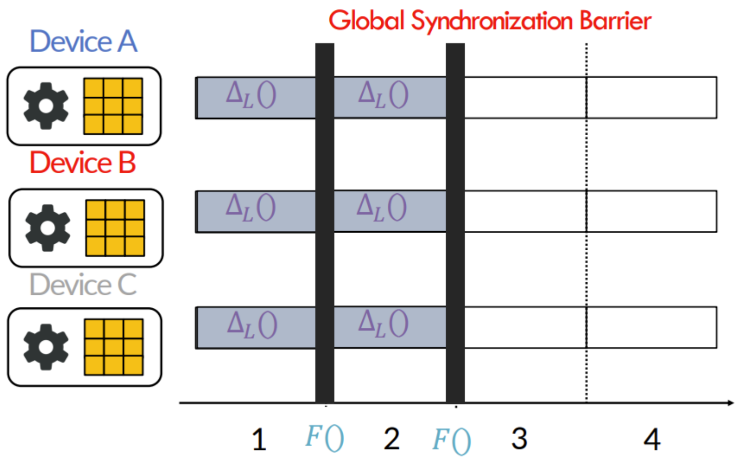


Figure 9: Idea case for distributed gradient descent

The problem of stragglers

Binary Synchronous Parallel (BSP) computing suffers from the issue of stragglers, which can significantly impact performance.

- Slow devices, known as stragglers, can cause all devices to wait, leading to delays in computation.
- With more devices, the likelihood of encountering a straggler increases.
- Stragglers are often transient, resulting from temporary compute or network load in multi-user environments, or fluctuating environmental conditions such as temperature or vibrations.
- In large clusters or cloud environments, where stragglers are unavoidable, BSP's throughput is greatly decreased.

Efforts to mitigate the impact of stragglers are crucial for improving the efficiency and scalability of BSP-based computing systems.

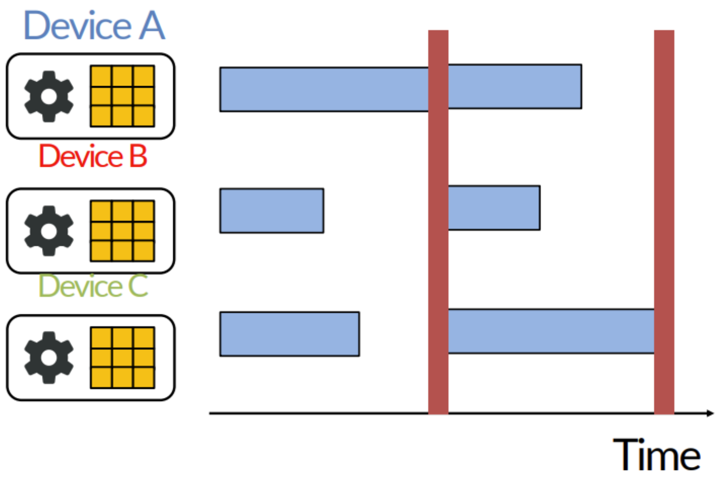


Figure 10: stragglers

Utilize the interesting property of Gradient Descent:

Machine learning (ML) is error-tolerant due to its inherent ability to handle and adapt to imperfect or noisy data. This characteristic allows ML models to produce useful outputs even when confronted with incomplete, inaccurate, or uncertain information. It also holds for gradient descent algorithms.

Background: Asynchronous Communication (No Consistency)

Asynchronous communication (Async) removes all communication barriers, maximizing computing time. However, it comes with its own set of challenges:

- Transient stragglers can cause messages to become extremely stale. For instance, Device 2 may be at time $t = 6$, but Device 1 has only sent a message for time $t = 1$.
- Some Async software allows messages to be applied while computing functions ($F()$) or updates ($\Delta L()$), leading to unpredictable behavior that can potentially hurt statistical efficiency.

While asynchronous communication offers advantages in terms of maximizing computing time, its lack of consistency can introduce uncertainties and challenges, particularly in distributed computing environments.

Background: Strict Consistency

Strict consistency, often seen in systems such as Bulk Synchronous Parallel (BSP), forms the baseline for many distributed machine learning frameworks including MapReduce, Spark, and various distributed machine learning (DistML) systems. In this approach:

- Devices compute updates $\Delta L()$ between global barriers, which serve as iteration boundaries.

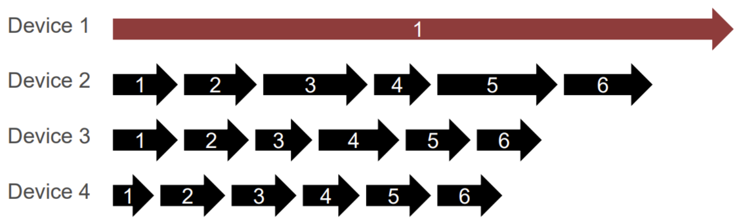


Figure 11: Asynchronous Communication

- Messages $\mathcal{M}$ are exchanged only during these barriers.
- The advantage of strict consistency lies in its serializability, offering the same guarantees as sequential algorithms.
- This consistency is provided that the aggregation function $F()$ is agnostic to the order of messages $\mathcal{M}$ (e.g., in Stochastic Gradient Descent).

Strict consistency ensures a predictable and reliable execution flow, crucial for maintaining accuracy and reproducibility in distributed machine learning systems.

Background: Bounded Consistency

Bounded consistency models, such as Stale Synchronous Parallel (SSP), offer a middle ground between Bulk Synchronous Parallel (BSP) and fully asynchronous (no-barrier) approaches. In SSP:

- Devices are allowed to iterate at different speeds.
- The fastest and slowest devices must not drift more than s iterations apart, where s represents the maximum staleness.
- For example, in SSP with a maximum staleness of $s = 3$, the fastest and slowest devices are allowed to diverge in their iterations, but the difference must not exceed s iterations.

Bounded consistency models like SSP balance the trade-offs between synchronization overhead and computation efficiency, making them suitable for various distributed computing tasks.

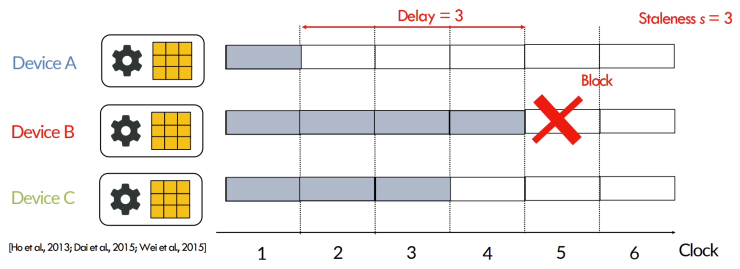


Figure 12: Bounded Consistency

Stale Synchronous Parallel (SSP): ”Lazy” Communication

In Stale Synchronous Parallel (SSP), devices prioritize efficiency by avoiding communication unless absolutely necessary. Specifically:

- Devices communicate only when the staleness condition is on the verge of being violated.
- This approach prioritizes throughput, aiming to maximize computational efficiency, even if it comes at the expense of statistical efficiency.

SSP’s ”lazy” communication strategy strikes a balance between synchronization overhead and computational efficiency in distributed computing environments. Here is a graph that shows the training loss under different

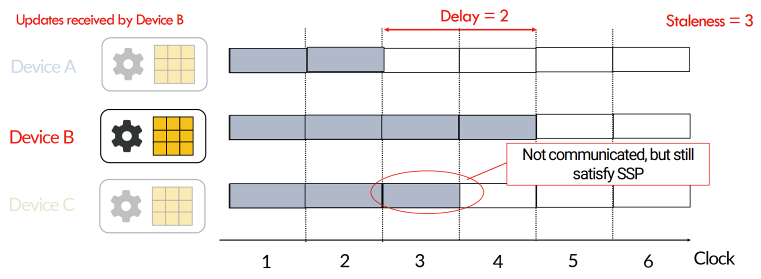

Updates received by Device B
Delay = 2
Staleness = 3
Device A
Device B
Device C
Not communicated, but still satisfy SSP
1
2
3
4
5
6
Clock

Figure 13: Stale Synchronous Parallel (SSP): ”Lazy” Communication

delays. We can see that the training loss diverges under high delay.

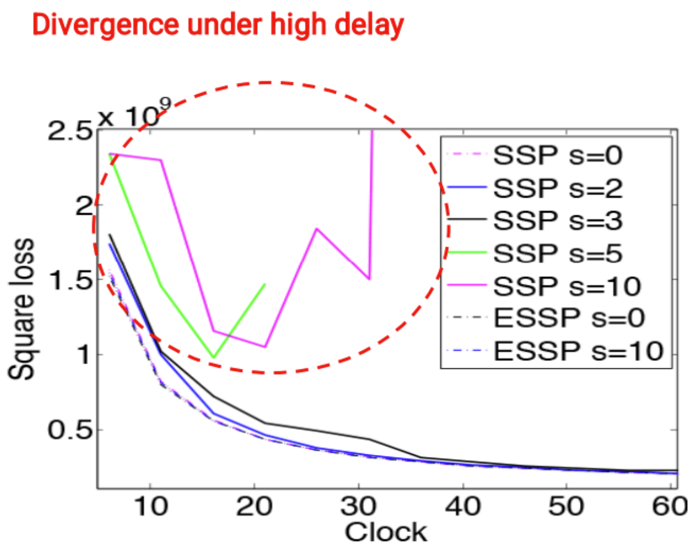


Figure 14: Impacts of Consistency/Staleness: Unbounded Staleness