

Architecture of ML Systems

05 Data- and Task-Parallel Execution

Matthias Boehm

Graz University of Technology, Austria
Computer Science and Biomedical Engineering
Institute of Interactive Systems and Data Science
BMVIT endowed chair for Data Management

Last update: Apr 03, 2020

Announcements/Org

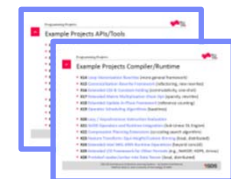
■ #1 Video Recording

- Link in **TeachCenter** & **TUbe** (lectures will be public)
- Streaming: <https://tugraz.webex.com/meet/m.boehm>
- Online teaching **extended until Apr 30**



■ #2 AMLS Programming Projects

- **Project selection** by **Apr 03** (see **Lecture 02/04**)
- **Status:** So far, project discussions w/ **11 students**
- There is a project for everybody, no matter the background
- Grace period until **April 17** (email to m.boehm@tugraz.at)



■ #3 Update Apache SystemDS

- **SystemDS merged**, in process of name change
- Already some bug reports by students (e.g., [SYSTEMML-2538](#))
- (Build and debug tutorial after Eastern)



Agenda

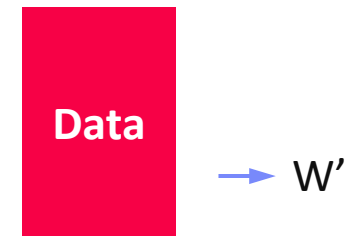
- **Motivation and Terminology**
- **Background MapReduce and Spark**
- **Data-Parallel Execution**
- **Task-Parallel Execution**

Motivation and Terminology

Terminology Batch/Mini-batch

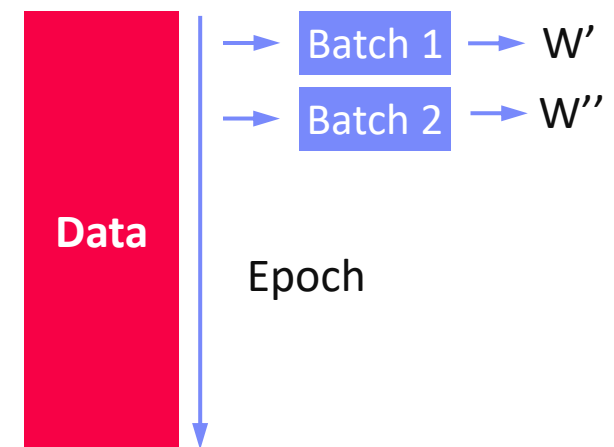
■ Batch ML Algorithms

- Iterative ML algorithms, where each iteration uses the **entire dataset** to compute gradients ΔW
- For (pseudo-) **second-order methods**, many features
- **Dedicated optimizers** for traditional ML algorithms



■ Mini-batch ML Algorithms

- Iterative ML algorithms, where each iteration only uses a **batch of rows** to make the next model update (in **epochs** or w/ **sampling**)
- For large and **highly redundant training sets**
- **Applies to almost all iterative**, model-based ML algorithms (LDA, reg., class., factor., DNN)
- **Stochastic Gradient Descent** (SGD)



Terminology Batch/Mini-batch, cont.

■ Excursus: Mini-batch Sizes



Yann LeCun @ylecun · 26. Apr. 2018

Training with large minibatches is bad for your health.
More importantly, it's bad for your test error.

Friends dont let friends use minibatches larger than 32. arxiv.org/abs/1804.07612



22



461



1.306



[Dominic Masters, Carlo
Luschi: Revisiting Small Batch
Training for Deep Neural
Networks. **CoRR 2018**]



- **Beware:** increased variance, HW vs statistical efficiency, data and model complexity, as well as used optimizer and parameters

■ Excursus: Pronunciation

- Stochastic Gradient **Descent**, not **decent**

Terminology Parallelism

■ Flynn's Classification

- SISD, SIMD
- (MISD), MIMD



[Michael J. Flynn, Kevin W. Rudd: Parallel Architectures. ACM Comput. Surv. 28(1) 1996]

Single
Instruction

Single Data

SISD
(uni-core)

Multiple Data

SIMD
(vector)

Multiple
Instruction

MISD
(pipelining)

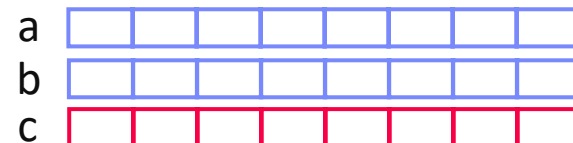
MIMD
(multi-core)

■ Example: SIMD Processing

- Streaming SIMD Extensions (SSE)
- Process the same operation on multiple elements at a time
(**packed** vs scalar SSE instructions)
- **Data parallelism**
(aka: instruction-level parallelism)
- Example: **VFMADD132PD**

2009 Nehalem: **128b** (2xFP64)
2012 Sandy Bridge: **256b** (4xFP64)
2017 Skylake: **512b** (8xFP64)

```
c = _mm512_fmadd_pd(a, b);
```



Recap: Central Data Abstractions

■ #1 Files and Objects

- **File:** Arbitrarily large sequential data in specific file format (CSV, binary, etc)
- **Object:** binary large object, with certain meta data

■ #2 Distributed Collections

- Logical multi-set (**bag**) of **key-value pairs** (**unsorted collection**)
- Different physical representations
- **Easy distribution** of pairs via horizontal partitioning (aka shards, partitions)
- Can be created from single file, or directory of files (unsorted)

Key	Value
4	Delta
2	Bravo
1	Alpha
3	Charlie
5	Echo
6	Foxtrott
7	Golf

Recap: Fault Tolerance & Resilience

[Google Data Center:

<https://www.youtube.com/watch?v=XZmGGAbHqa0>]

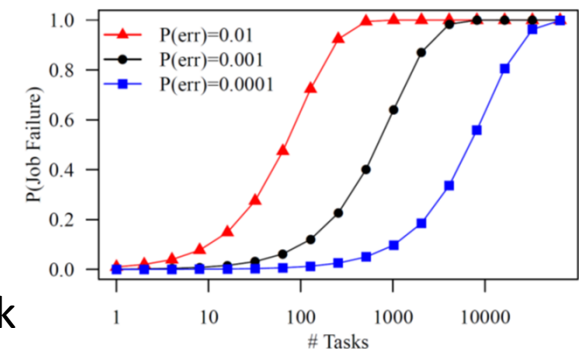
■ Resilience Problem

- Increasing error rates **at scale** (soft/hard mem/disk/net errors)
- Robustness for preemption
- **Need for cost-effective resilience**



■ Fault Tolerance in Large-Scale Computation

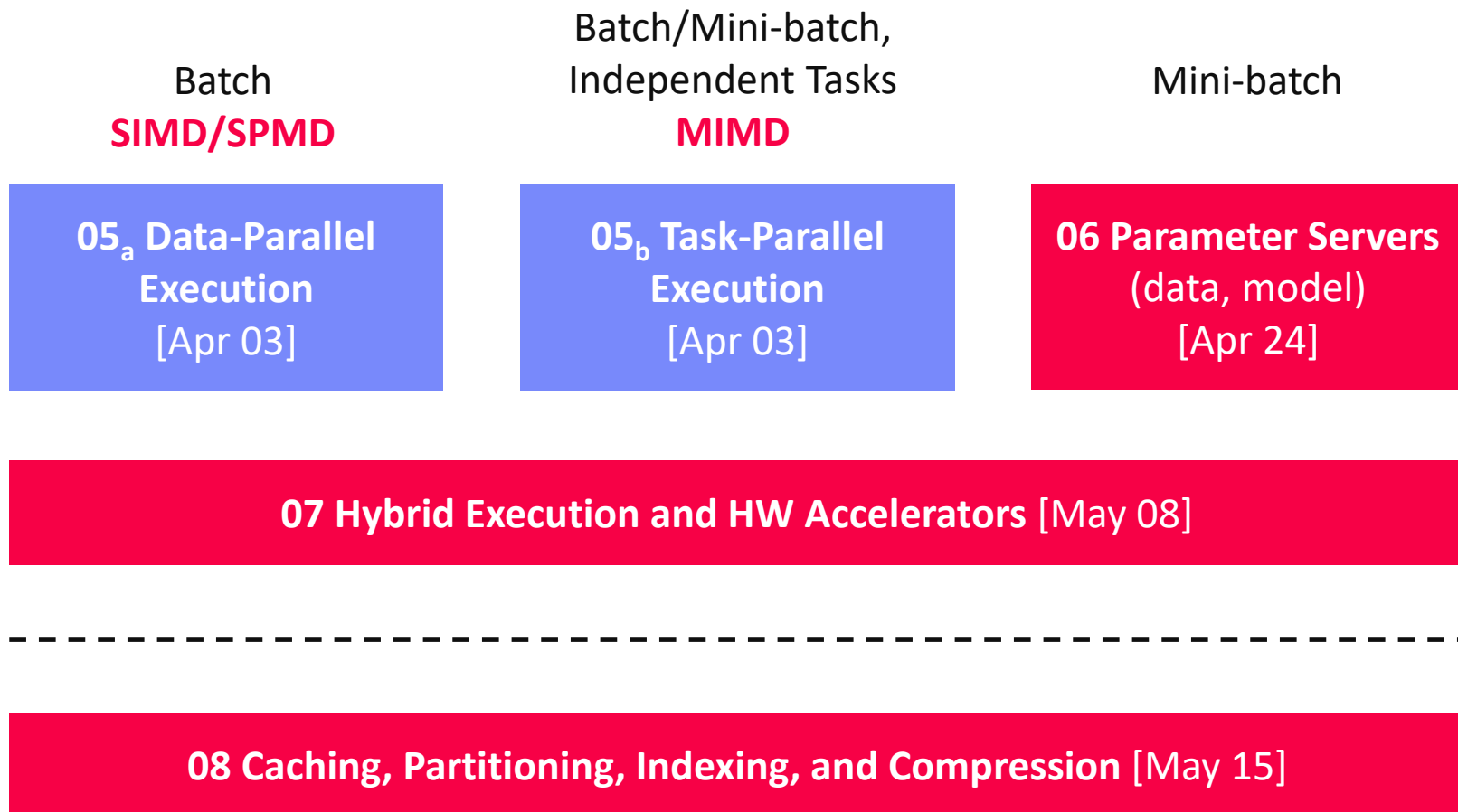
- Block replication in distributed file systems
- ECC; checksums for blocks, broadcast, shuffle
- Checkpointing (all task outputs / on request)
- Lineage-based recomputation for recovery in Spark



■ ML-specific Approaches (exploit app characteristics)

- Estimate contribution from lost partition to avoid stragglers
- Example: user-defined “compensation” functions

Categories of Execution Strategies



Background MapReduce and Spark (Data-Parallel Collection Processing)

Abstractions for Fault-tolerant,
Distributed Storage and Computation

Hadoop History and Architecture

■ Recap: Brief History

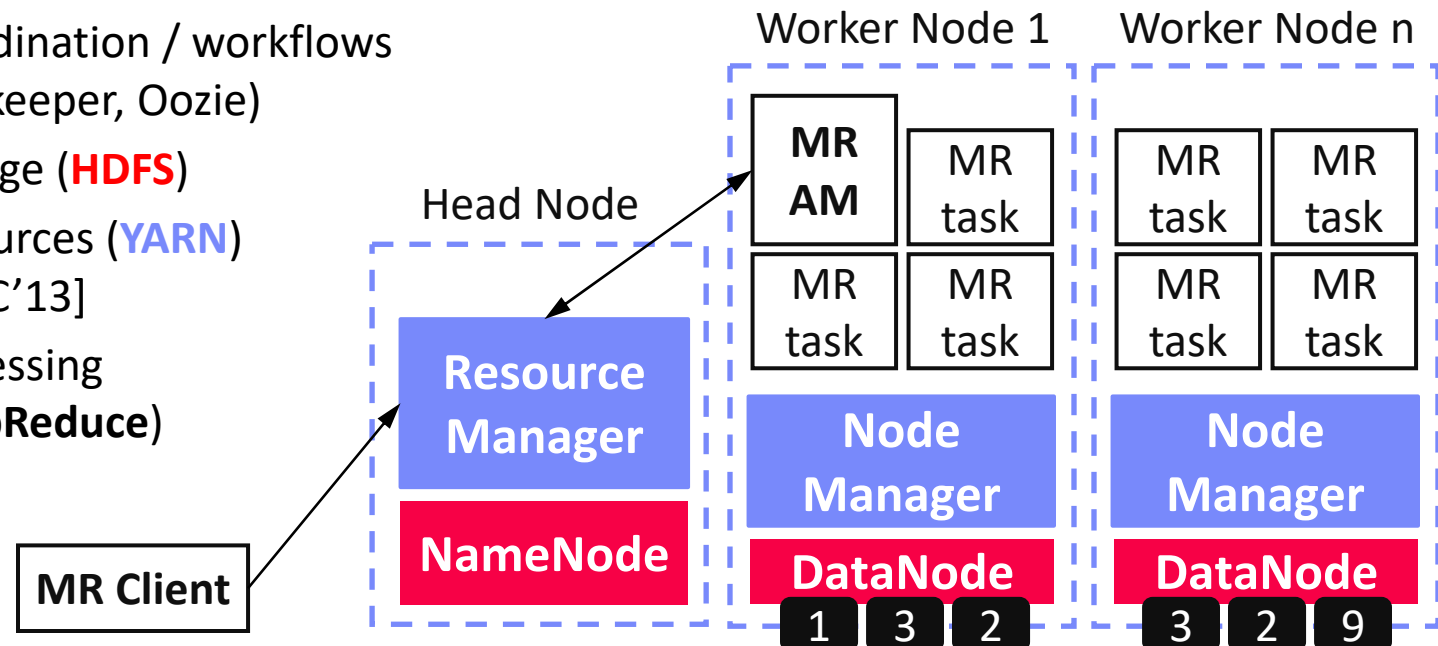
- Google's GFS [SOSP'03] + MapReduce
→ **Apache Hadoop** (2006)
- Apache Hive (SQL), Pig (ETL), Mahout (ML), Giraph (Graph)

[Jeffrey Dean, Sanjay
Ghemawat: MapReduce:
Simplified Data Processing on
Large Clusters. **OSDI 2004**]



■ Hadoop Architecture / Eco System

- Management (Ambari)
- Coordination / workflows (Zookeeper, Oozie)
- Storage (**HDFS**)
- Resources (**YARN**) [SoCC'13]
- Processing (MapReduce)



MapReduce – Programming Model

Overview Programming Model

- Inspired by functional programming languages
- **Implicit parallelism** (abstracts distributed storage and processing)
- **Map** function: key/value pair → set of intermediate key/value pairs
- **Reduce** function: merge all intermediate values by key

Example `SELECT Dep, count(*) FROM csv_files GROUP BY Dep`

Name	Dep
X	CS
Y	CS
A	EE
Z	CS

Collection of
key/value pairs

```
map(Long pos, String line) {
  parts ← line.split(",")
  emit(parts[1], 1)
}
```

CS	1
CS	1
EE	1
CS	1

```
reduce(String dep,
  Iterator<Long> iter) {
  total ← iter.sum();
  emit(dep, total)
}
```

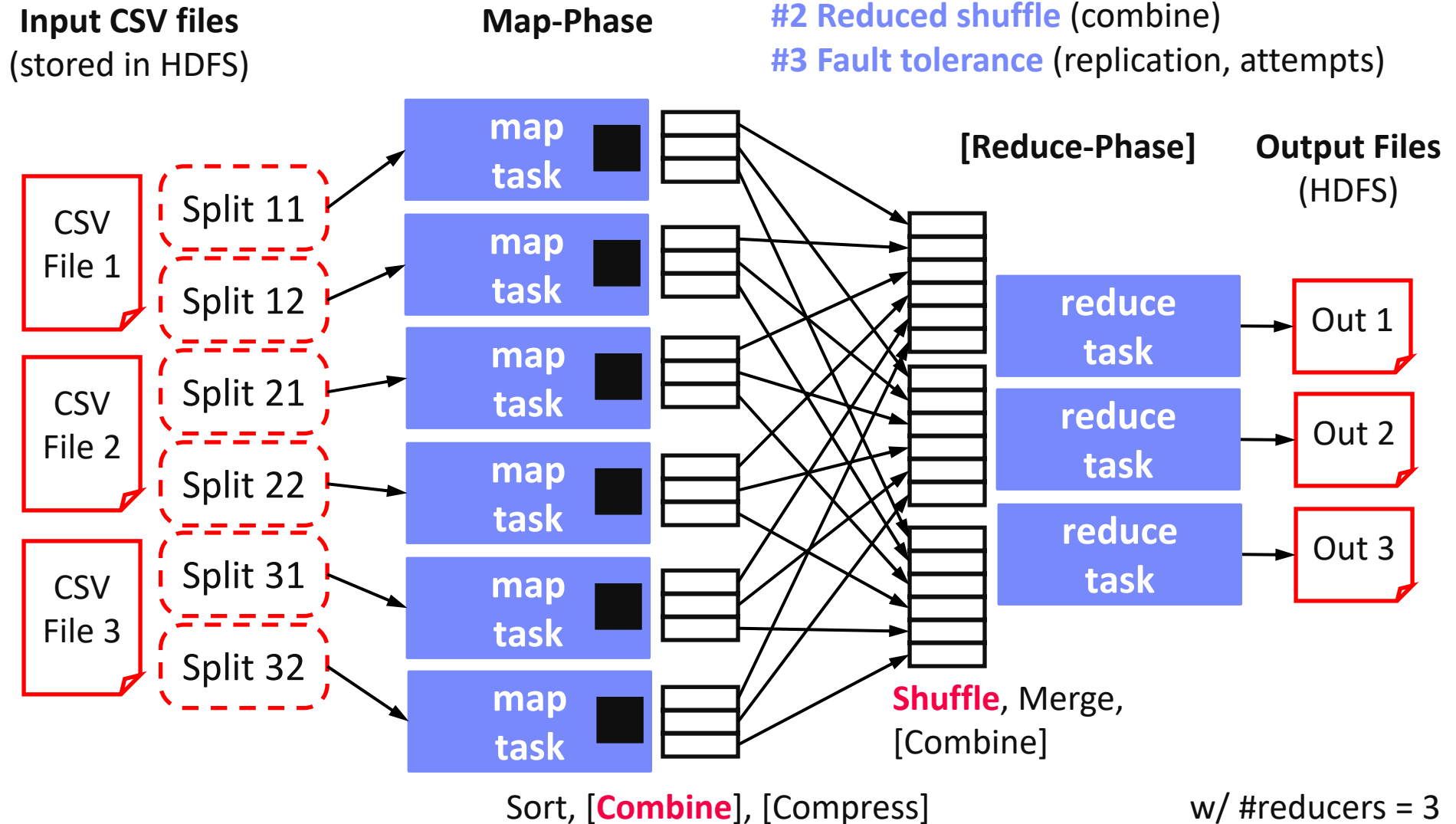
CS	3
EE	1

MapReduce – Execution Model

#1 **Data Locality** (delay sched., write affinity)

#2 **Reduced shuffle** (combine)

#3 **Fault tolerance** (replication, attempts)



Spark History and Architecture

■ Summary MapReduce

- Large-scale & fault-tolerant processing w/ UDFs and files → **Flexibility**
- Restricted functional APIs → **Implicit parallelism and fault tolerance**
- **Criticism: #1 Performance, #2 Low-level APIs, #3 Many different systems**

■ Evolution to Spark (and Flink)

- Spark [HotCloud'10] + RDDs [NSDI'12] → **Apache Spark** (2014)
- **Design:** **standing executors with in-memory storage**, lazy evaluation, and fault-tolerance via RDD lineage
- **Performance:** In-memory storage and fast job scheduling (100ms vs 10s)
- **APIs:** Richer functional APIs and general computation DAGs, high-level APIs (e.g., DataFrame/Dataset), unified platform



➔ But many shared concepts/infrastructure

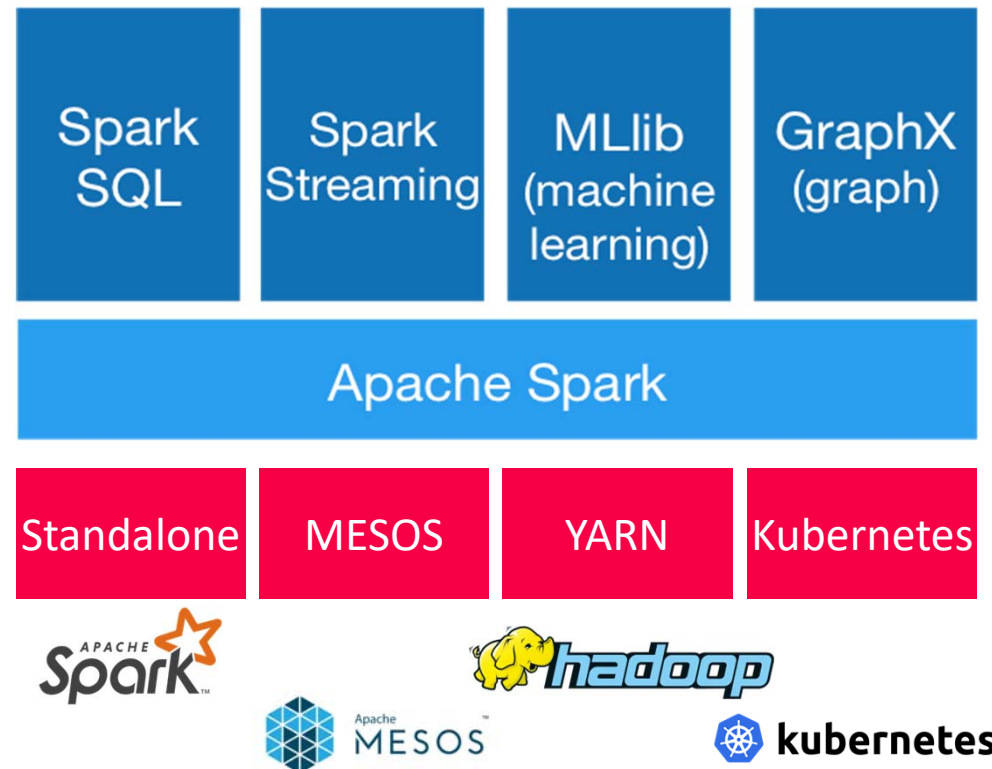
- **Implicit parallelism through dist. collections** (data access, fault tolerance)
- Resource negotiators (YARN, Mesos, Kubernetes)
- HDFS and object store connectors (e.g., Swift, S3)

Spark History and Architecture, cont.

■ High-Level Architecture

- **Different language bindings:**
Scala, Java, Python, R
- **Different libraries:**
SQL, ML, Stream, Graph
- Spark core (incl RDDs)
- **Different cluster managers:**
Standalone, Mesos, Yarn, Kubernetes
- Different file systems/
formats, and data sources:
HDFS, S3, SWIFT, DBs, NoSQL

[<https://spark.apache.org/>]



- Focus on a **unified** platform
for data-parallel computation (**Apache Flink** w/ similar goals)

Spark Resilient Distributed Datasets (RDDs)

■ RDD Abstraction

`JavaPairRDD<MatrixIndexes,MatrixBlock>`

- **Immutable**, partitioned
collections of key-value pairs
- **Coarse-grained** deterministic operations (transformations/actions)
- Fault tolerance via lineage-based re-computation

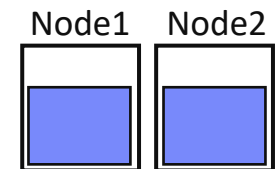
■ Operations

- Transformations: define new RDDs
- Actions: return result to driver

Type	Examples
Transformation (lazy)	<code>map</code> , <code>hadoopFile</code> , <code>textFile</code> , <code>flatMap</code> , <code>filter</code> , <code>sample</code> , <code>join</code> , <code>groupByKey</code> , <code>cogroup</code> , <code>reduceByKey</code> , <code>cross</code> , <code>sortByKey</code> , <code>mapValues</code>
Action	<code>reduce</code> , <code>save</code> , <code>collect</code> , <code>count</code> , <code>lookupKey</code>

■ Distributed Caching

- Use fraction of worker **memory for caching**
- Eviction at granularity of individual partitions
- **Different storage levels** (e.g., mem/disk x serialization x compression)

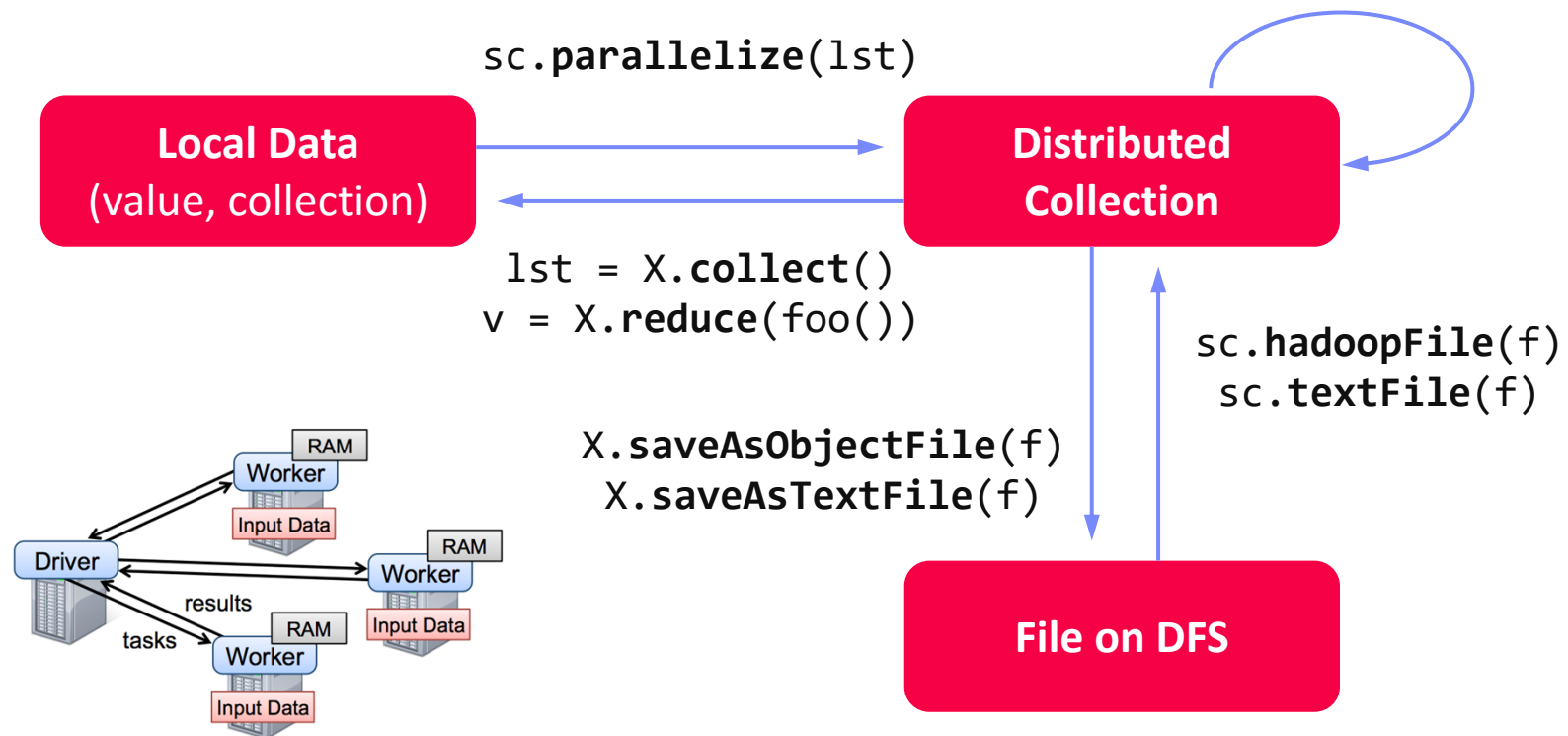


Spark Resilient Distributed Datasets (RDDs), cont.

■ Lifecycle of an RDD

- **Note:** can't broadcast an RDD directly

```
X.filter(foo())
X.mapValues(foo())
X.reduceByKey(foo())
X.cache()/X.persist(...)
```



Spark Partitions and Implicit/Explicit Partitioning

■ Spark Partitions

- Logical key-value collections are split into **physical partitions** ~128MB
- Partitions are granularity of **tasks, I/O, shuffling, evictions**

■ Partitioning via Partitioners

- Implicitly on every data shuffling
- Explicitly via `R.repartition(n)`

Example Hash Partitioning:

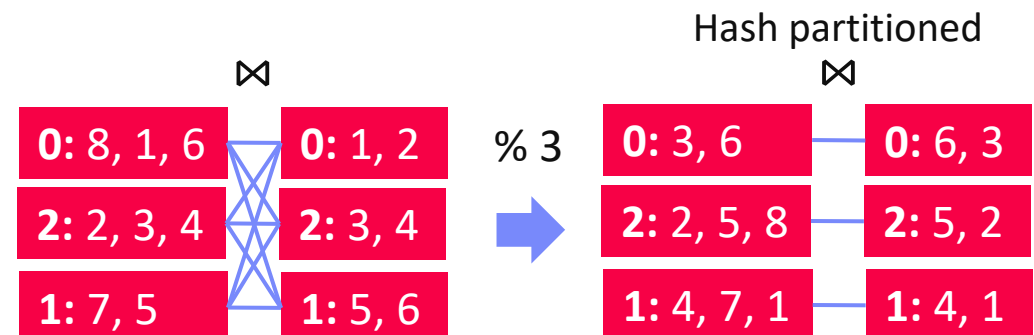
For all (k,v) of R:
 $\text{pid} = \text{hash}(k) \% n$

■ Partitioning-Preserving

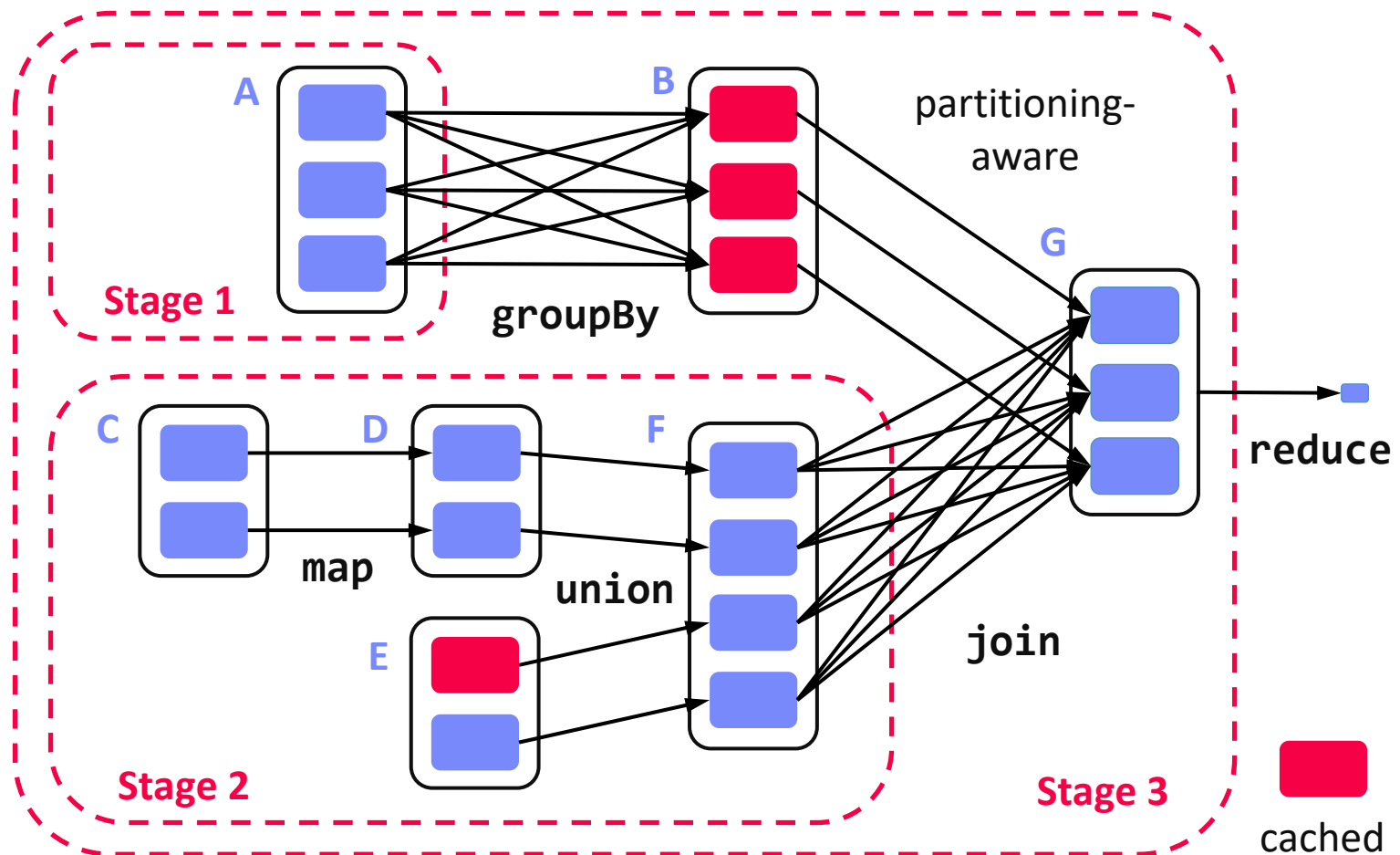
- All operations that are guaranteed to keep keys unchanged (e.g. `mapValues()`, `mapPartitions()` w/ `preservesPart` flag)

■ Partitioning-Exploiting

- Join: `R3 = R1.join(R2)`
- Lookups:
`v = C.lookup(k)`



Spark Lazy Evaluation, Caching, and Lineage



[Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, Ion Stoica: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. **NSDI 2012**]

Data-Parallel Execution

Batch ML Algorithms

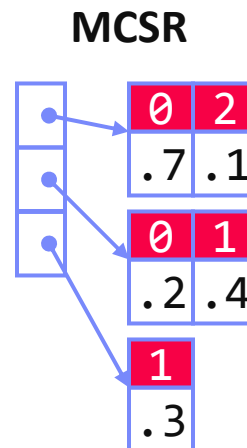
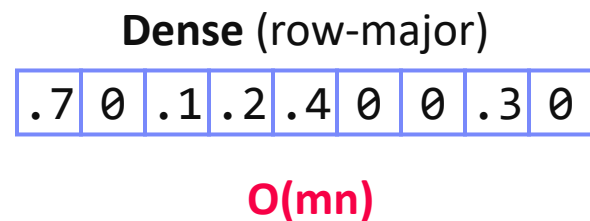


Background: Matrix Formats

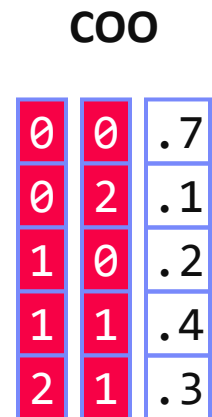
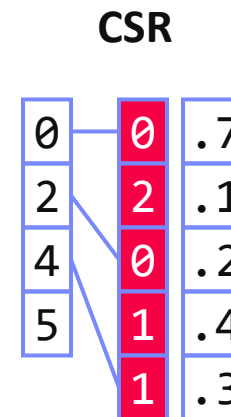
- **Matrix Block** ($m \times n$)
 - A.k.a. tiles/chunks, most operations defined here
 - Local matrix: single block, different representations
- **Common Block Representations**
 - Dense (linearized arrays)
 - MCSR (modified CSR)
 - CSR (compressed sparse rows), CSC
 - COO (Coordinate matrix)

Example
3x3 Matrix

.7		.1
.2	.4	
	.3	



$O(m + \text{nnz}(X))$



$O(\text{nnz}(X))$

Distributed Matrix Representations

Collection of “Matrix Blocks” (and keys)

- **Bag semantics** (duplicates, unordered)
- Logical (Fixed-Size) Blocking
+ **join processing / independence**
- **(sparsity skew)**
- E.g., SystemML on Spark:
`JavaPairRDD<MatrixIndexes, MatrixBlock>`
- Blocks encoded independently (dense/sparse)

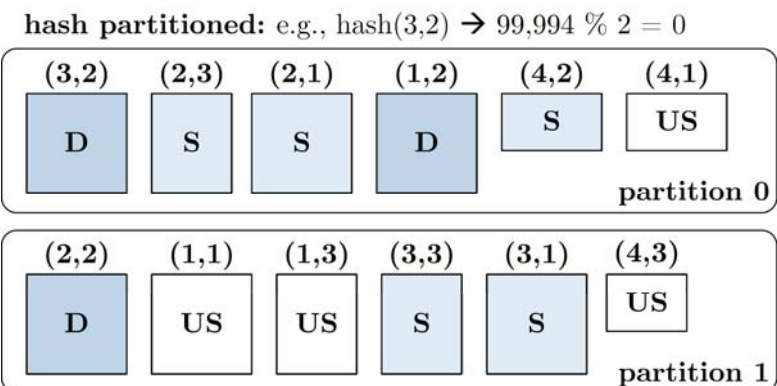
Logical Blocking
3,400x2,700 Matrix
(w/ $B_c=1,000$)

(1,1)	(1,2)	(1,3)
(2,1)	(2,2)	(2,3)
(3,1)	(3,2)	(3,3)
(4,1)	(4,2)	(4,3)

Partitioning

- Logical Partitioning
(e.g., row-/column-wise)
- Physical Partitioning
(e.g., hash / grid)

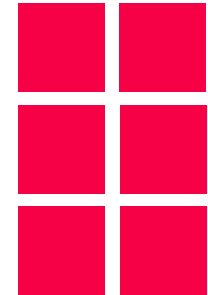
Physical
Blocking and
Partitioning



Distributed Matrix Representations, cont.

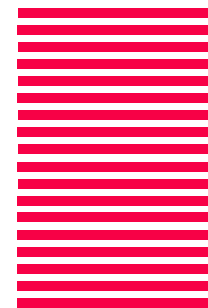
■ #1 Block-partitioned Matrices

- Fixed-size, square or rectangular blocks
- **Pros:** Input/output alignment, block-local transpose, amortize block overheads, bounded mem, cache-conscious
- **Cons:** Converting row-wise inputs (e.g., text) requires shuffle
- **Examples:** RIOT, PEGASUS, SystemML, SciDB, Cumulon, Distributed R, DMac, Spark MLlib, Gilbert, MatFast, and SimSQL



■ #2 Row/Column-partitioned Matrices

- Collection of row indexes and rows (or columns respectively)
- **Pros:** Seamless data conversion and access to entire rows
- **Cons:** Storage overhead in Java, and cache unfriendly operations
- **Examples:** Spark MLlib, Mahout Samsara, Emma, SimSQL



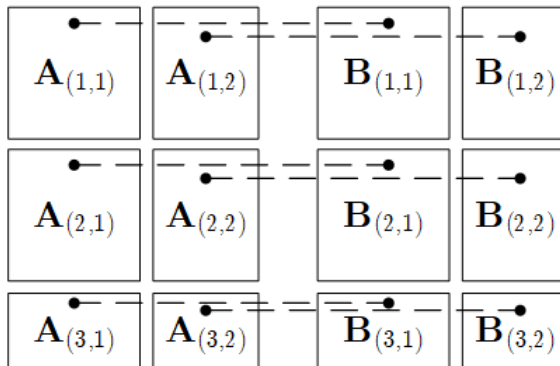
■ #3 Algorithm-specific Partitioning

- Operation and algorithm-centric data representations
- **Examples:** matrix **inverse**, matrix **factorization**

Distributed Matrix Operations

Elementwise Multiplication (Hadamard Product)

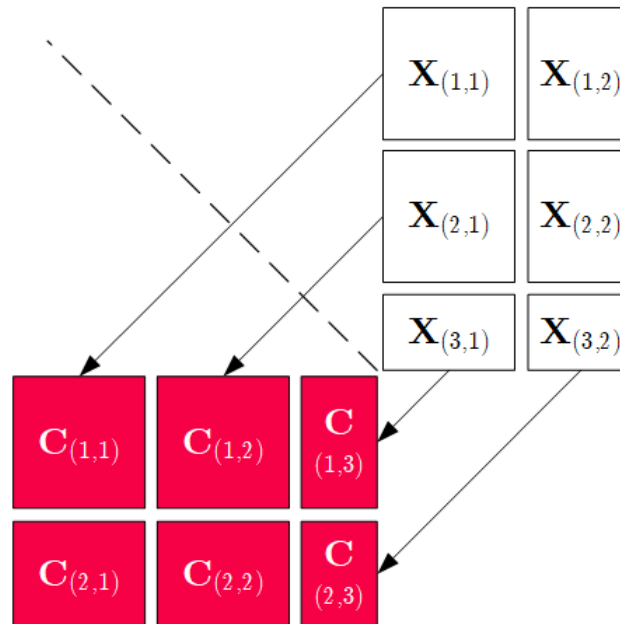
$$C = A * B$$



Note: also with
row/column vector rhs

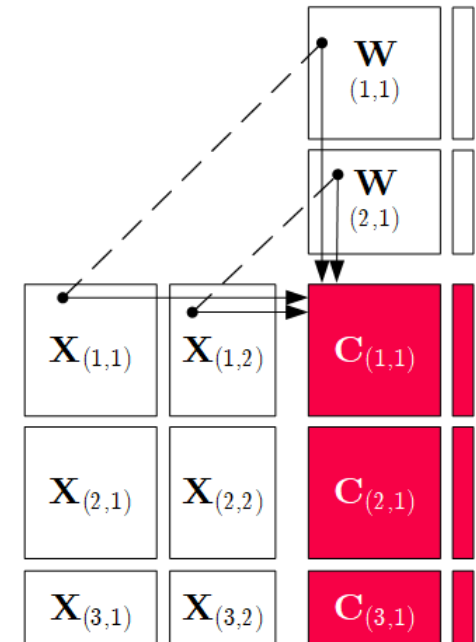
Transposition

$$C = t(X)$$



Matrix Multiplication

$$C = X \%* \% W$$



Note: 1:N join

Physical MM Operator Selection

Common Selection Criteria

- **Data and cluster characteristics** (e.g., data size/shape, memory, parallelism)
- **Matrix/operation properties** (e.g., diagonal/symmetric, sparse-safe ops)
- **Data flow properties** (e.g., co-partitioning, co-location, data locality)

#0 Local Operators

- SystemML `mm`, `tsmm`, `mmchain`; Samsara/Mllib local

#1 Special Operators (special patterns/sparsity)

- SystemML `tsmm`, `mapmmchain`; Samsara AtA

#2 Broadcast-Based Operators (aka broadcast join)

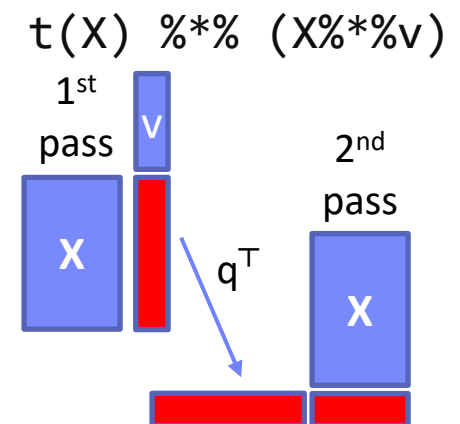
- SystemML `mapmm`, `mapmmchain`

#3 Co-Partitioning-Based Operators (aka improved repartition join)

- SystemML `zipmm`; Emma, Samsara OpAtB

#4 Shuffle-Based Operators (aka repartition join)

- SystemML `cpmm`, `rmm`; Samsara OpAB



Partitioning-Preserving Operations

- **Shuffle is major bottleneck** for ML on Spark
- **Preserve Partitioning**
 - Op is partitioning-preserving if keys unchanged (guaranteed)
 - Implicit: Use restrictive APIs (`mapValues()` vs `mapToPair()`)
 - Explicit: Partition computation w/ declaration of partitioning-preserving
- **Exploit Partitioning**
 - Implicit: Operations based on `join`, `cogroup`, etc
 - Explicit: Custom operators (e.g., `zipmm`)

- **Example:**
Multiclass SVM

- Vectors fit neither into driver nor broadcast
- $\text{ncol}(X) \leq B_c$

```

parfor(iter_class in 1:num_classes) {
  Y_local = 2 * (Y == iter_class) - 1
  g_old = t(X) %*% Y_local
  ...
  while( continue ) {
    Xd = X %*% s
    ... inner while loop (compute step_sz)
    Xw = Xw + step_sz * Xd;
    out = 1 - Y_local * Xw;
    out = (out > 0) * out;
    g_new = t(X) %*% (out * Y_local) ...
  }
}

```

Annotations:

- `parfor(iter_class in 1:num_classes)` ← **repart, chkpt X MEM_DISK**
- `Y_local = 2 * (Y == iter_class) - 1` ← **chkpt y_local MEM_DISK**
- `Xd = X %*% s` ← **chkpt Xd, Xw MEM_DISK**
- `g_new = t(X) %*% (out * Y_local) ...` ← **zipmm**

Dask DASK

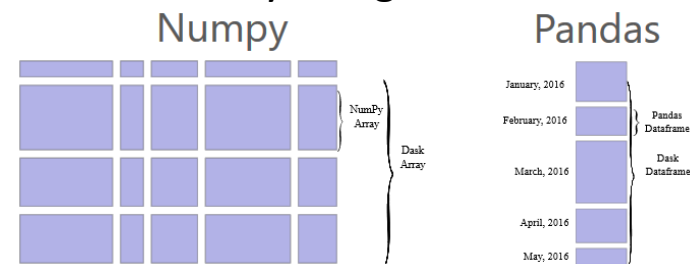
[Matthew Rocklin: Dask: Parallel Computation with Blocked algorithms and Task Scheduling, **Python in Science 2015**]

[Dask Development Team: Dask: Library for dynamic task scheduling, 2016, <https://dask.org>]



■ Overview Dask

- Multi-threaded and distributed operations for arrays, bags, and dataframes
- dask.array:**
list of numpy n-dim arrays
- dask.dataframe:**
list of pandas data frames
- dask.bag:** unordered list of tuples (second order functions)
- Local and distributed schedulers:
threads, processes, YARN, Kubernetes, containers, HPC, and cloud, GPUs



■ Execution

- Lazy evaluation**
- Limitation: requires **static size inference**
- Triggered via `compute()`

```
import dask.array as da

x = da.random.random(
    (10000,10000), chunks=(1000,1000))
y = x + x.T
y.persist() # cache in memory
z = y[:,::2, 5000:].mean(axis=1)
ret = z.compute() # returns NumPy array
```


Task-Parallel Execution

Parallel Computation of Independent Tasks,
Emulation of Data-Parallel Operations/Programs



Overview Task-Parallelism

■ Historic Perspective

- Since 1980s: various parallel Fortran extensions, especially in HPC
- **DOALL parallel loops** (independent iterations)
- OpenMP (since 1997, Open Multi-Processing)


```
#pragma omp parallel for reduction(+: nnz)
for (int i = 0; i < N; i++) {
    int threadID = omp_get_thread_num();
    R[i] = foo(A[i]);
    nnz += (R[i]!=0) ? 1 : 0;
}
```

■ Motivation: Independent Tasks in ML Workloads

- **Use cases:** Ensemble learning, cross validation, hyper-parameter tuning, complex models with disjoint/overlapping/all data per task
- **Challenge #1:** Adaptation to data and cluster characteristics
- **Challenge #2:** Combination with data-parallelism

Parallel For Loops (ParFor)

[M. Boehm et al.: Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. **PVLDB 2014**]



Hybrid Parallelization Strategies

- Combination of **data- and task-parallel** ops
- Combination of **local and distributed** computation

Key Aspects

- Dependency Analysis
- Task partitioning
- Data partitioning, scan sharing, various rewrites
- Execution strategies
- Result agg strategies
- ParFor optimizer**

```
reg = 10^(seq(-1, -10))
B_all = matrix(0, nrow(reg), n)
```

```
parfor( i in 1:nrow(reg) ) {
  B = lm(X, y, reg[i,1]);
  B_all[i,] = t(B);
}
```

Local ParFor
(multi-threaded),
w/ local ops

Remote ParFor
(distributed
Spark job)

Local ParFor,
w/ concurrent
distributed ops

Additional ParFor Examples



■ Pairwise Pearson Correlation

- In practice: uni/bivariate stats
- Pearson's R, Anova F, Chi-squared, Degree of freedom, P-value, Cramers V, Spearman, etc)

```
D = read("./input/D");
R = matrix(0, ncol(D), ncol(D));
parfor(i in 1:(ncol(D)-1)) {
  X = D[,i];
  sX = sd(X);
  parfor(j in (i+1):ncol(D)) {
    Y = D[,j];
    sY = sd(Y);
    R[i,j] = cov(X,Y)/(sX*sY);
  }
}
write(R, "./output/R");
```

■ Batch-wise CNN Scoring

- Emulate data-parallelism for complex functions

```
prob = matrix(0, Ni, Nc)
parfor( i in 1:ceil(Ni/B) ) {
  Xb = X[((i-1)*B+1):min(i*B,Ni),];
  prob[((i-1)*B+1):min(i*B,Ni),] =
    ... # CNN scoring
}
```

➔ Conceptual Design:

Master/worker (task: group of parfor iterations)

ParFor Execution Strategies

#1 Task Partitioning

- Fixed-size schemes:
naive (1), static (n/k), fixed (m)
- Self-scheduling: e.g.,
guided self scheduling, factoring

Factoring (n=101, k=4)

$$R_0 = N, \quad R_{i+1} = R_i - k \cdot l_i, \quad l_i = \left\lceil \frac{R_i}{x_i \cdot k} \right\rceil = \left\lceil \left(\frac{1}{x_i} \right)^{i+1} \frac{N}{k} \right\rceil$$

(13,13,13,13, 7,7,7,7, 3,3,3,3, 2,2,2,2, 1)

#2 Data Partitioning

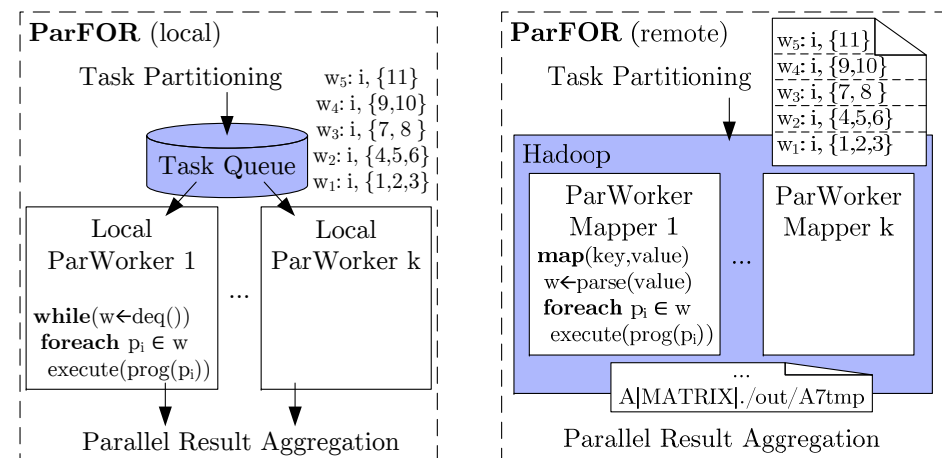
- Local or remote row/column partitioning (incl locality)

#3 Task Execution

- Local (multi-core) execution
- Remote (MR/Spark) execution

#4 Result Aggregation

- With and without compare (non-empty output variable)
- Local in-memory / remote MR/Spark result aggregation

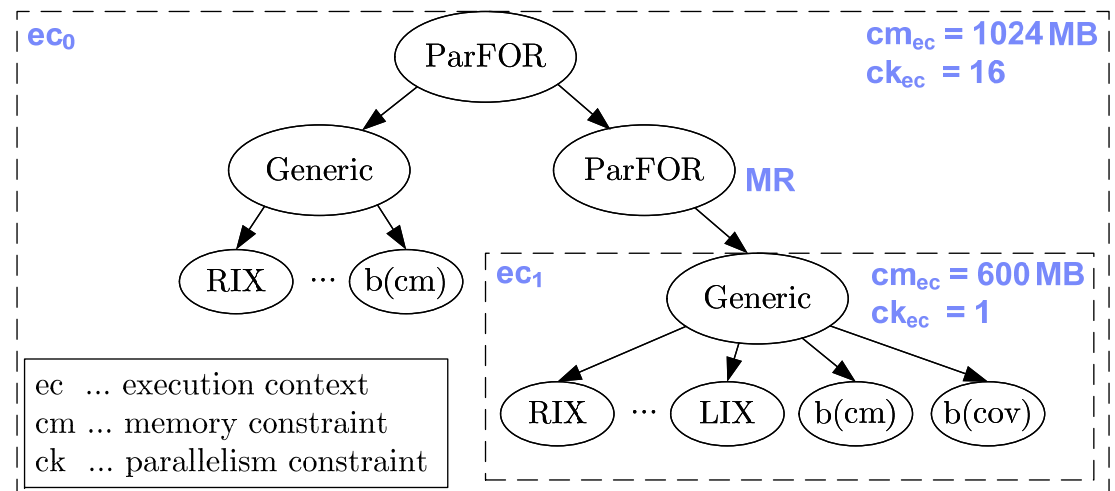


ParFor Optimizer Framework

- **Design:** Runtime optimization for each top-level parfor

- **Plan Tree P**

- Nodes N_p
 - Exec type et
 - Parallelism k
 - Attributes A
 - Height h
 - Exec contexts EC_p



- **Plan Tree Optimization Objective**

$$\phi_2 : \min \hat{T}(r(P))$$

$$s.t. \quad \forall ec \in \mathcal{EC}_P : \hat{M}(r(ec)) \leq cm_{ec} \wedge K(r(ec)) \leq ck_{ec}.$$

- **Heuristic optimizer w/ transformation-based search strategy**
 - Cost and memory estimates w/ plan tree aggregate statistics

Task-Parallelism in R



■ Multi-Threading

- **doMC** as multi-threaded foreach backend
- Foreach w/ parallel (%dopar%) or sequential (%do%) execution

[<https://cran.r-project.org/web/packages/doMC/vignettes/gettingstartedMC.pdf>]

```
library(doMC)
registerDoMC(32)
R <- foreach(i=1:(ncol(D)-1),
              .combine=rbind) %dopar% {
  X = D[,i]; sX = sd(X);
  Ri = matrix(0, 1, ncol(D))
  for(j in (i+1):ncol(D)) {
    Y = D[,j]; sY = sd(Y)
    Ri[1,j] = cov(X,Y)/(sX*sY);
  }
  return(Ri);
}
```

■ Distribution

- **doSNOW** as distributed foreach backend
- MPI/SOCK as comm methods

[<https://cran.r-project.org/web/packages/doSNOW/doSNOW.pdf>]

```
library(doSNOW)
clust = makeCluster(
  c("192.168.0.1", "192.168.0.2",
    "192.168.0.3"), type="SOCK");
registerDoSNOW(clust);
... %dopar% ...
stopCluster(clust);
```

Task-Parallelism in Other Systems

■ MATLAB

- Parfor loops for multi-process & distributed loops
- Use-defined par

```
matlabpool 32
c = pi; z = 0;
r = rand(1,10)
parfor i = 1 : 10
    z = z+1; # reduction
    b(i) = r(i); # sliced
end
```



[Gaurav Sharma, Jos Martin:
MATLAB®: A Language for
Parallel Computing. Int. **Journal**
on Parallel Prog. 2009]



■ Julia

- Dedicated macros:
@threads
@distributed

```
a = zeros(1000)
@threads for i in 1:1000
    a[i] = rand(r[threadid()])
end
```



[<https://docs.julialang.org/en/v1/manual/parallel-computing/>]

■ TensorFlow

- User-defined parallel iterations, responsible for correct results or acceptable approximate results

```
tf.nn.parallel_loop_run(cond, body, loop_vars, parallel_iterations=10,
    swap_memory=False, maximum_iterations=None, ...)
```



[https://www.tensorflow.org/api_docs/python/tf/while_loop]

Summary and Q&A

- **Categories of Execution Strategies**
 - **Data-parallel execution** for batch ML algorithms
 - **Task-parallel execution** for custom parallelization of independent tasks
 - Parameter servers (data-parallel vs model-parallel) for mini-batch ML algorithms
- **#1 Different strategies (and systems) for different ML workloads**
→ **Specialization and abstraction**
- **#2 Awareness of underlying execution frameworks**
- **#3 Awareness of effective compilation and runtime techniques**
- **Next Lectures (after Easter Break)**
 - **06 Parameter Servers** [Apr 24]
 - **07 Hybrid Execution and HW Accelerators** [May 08]
 - **08 Caching, Partitioning, Indexing and Compression** [May 15]