Concepts and Models of Parallel and Data-centric Programming

MapReduce – Yet Another Resource Negotiator

Lecture, Summer 2020

Simon Schwitanski
Dr. Christian Terboven

High Performance Computing

RWTH AACHEN UNIVERSITY

Outline

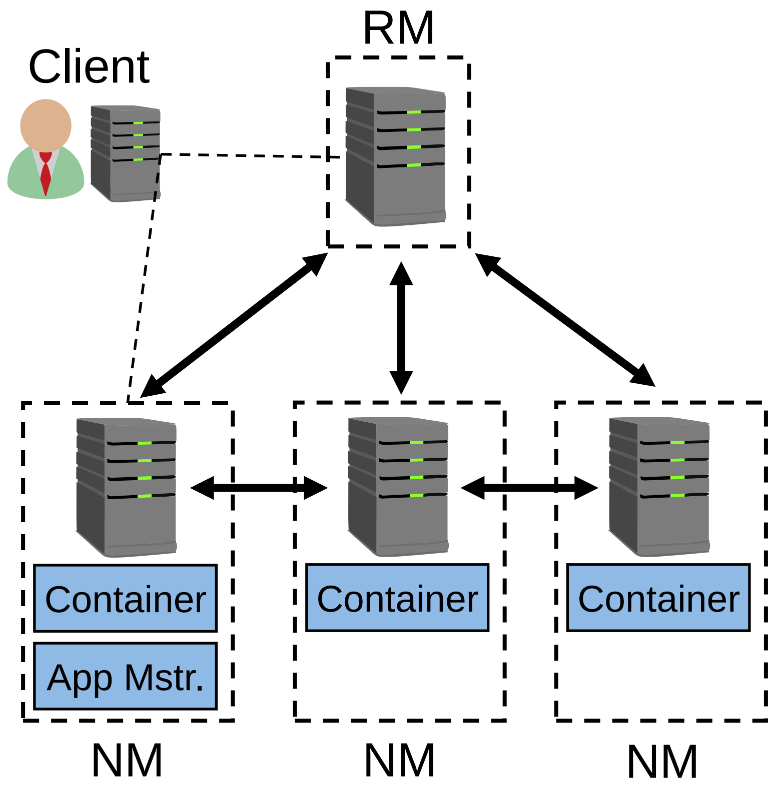
Yet Another Resource Negotiator

- Cluster resource management system of Hadoop
- Introduced in Hadoop 2, encapsulates resource management
- General enough to support other computing paradigms
- YARN API: Request and run applications on cluster resources
  - Typically, user gets not directly in contact with YARN API
  - Higher-layer frameworks (e.g., MapReduce) used

Application	MapReduce
Compute	Yet Another Resource Negotiator (YARN)
Storage	Hadoop Distributed Filesystem (HDFS)

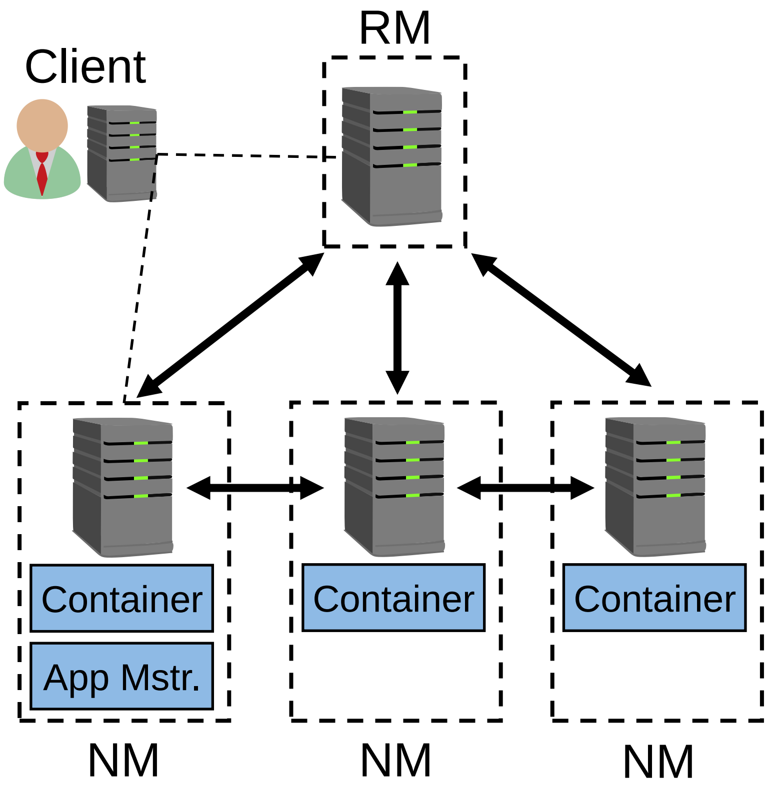
Core Components (1)

- Master / Worker architecture
- Resource Manager (RM)
  - Managing resources across whole cluster
  - One per cluster
  - Schedules jobs to NodeManager nodes
- Node Manager (NM)
  - Runs on each worker node
  - Responsible for running and monitoring *containers*
- Container
  - Represents allocated set of resources (memory, CPU, …) on a node
  - Application-specific task runs in container



Core Components (2)

- Application master
  - One instance per application
  - Started by RM, runs in a container on arbitrary NM
  - Negotiates further resources (i.e., further containers) with RM
  - Executes and monitors map and reduce tasks in containers on different NMs
  - Job coordination and tracking
- Client
  - Submits job to RM
  - RM creates application master on arbitrary NM, starts computation
  - Polls application master for job progress



YARN Architecture

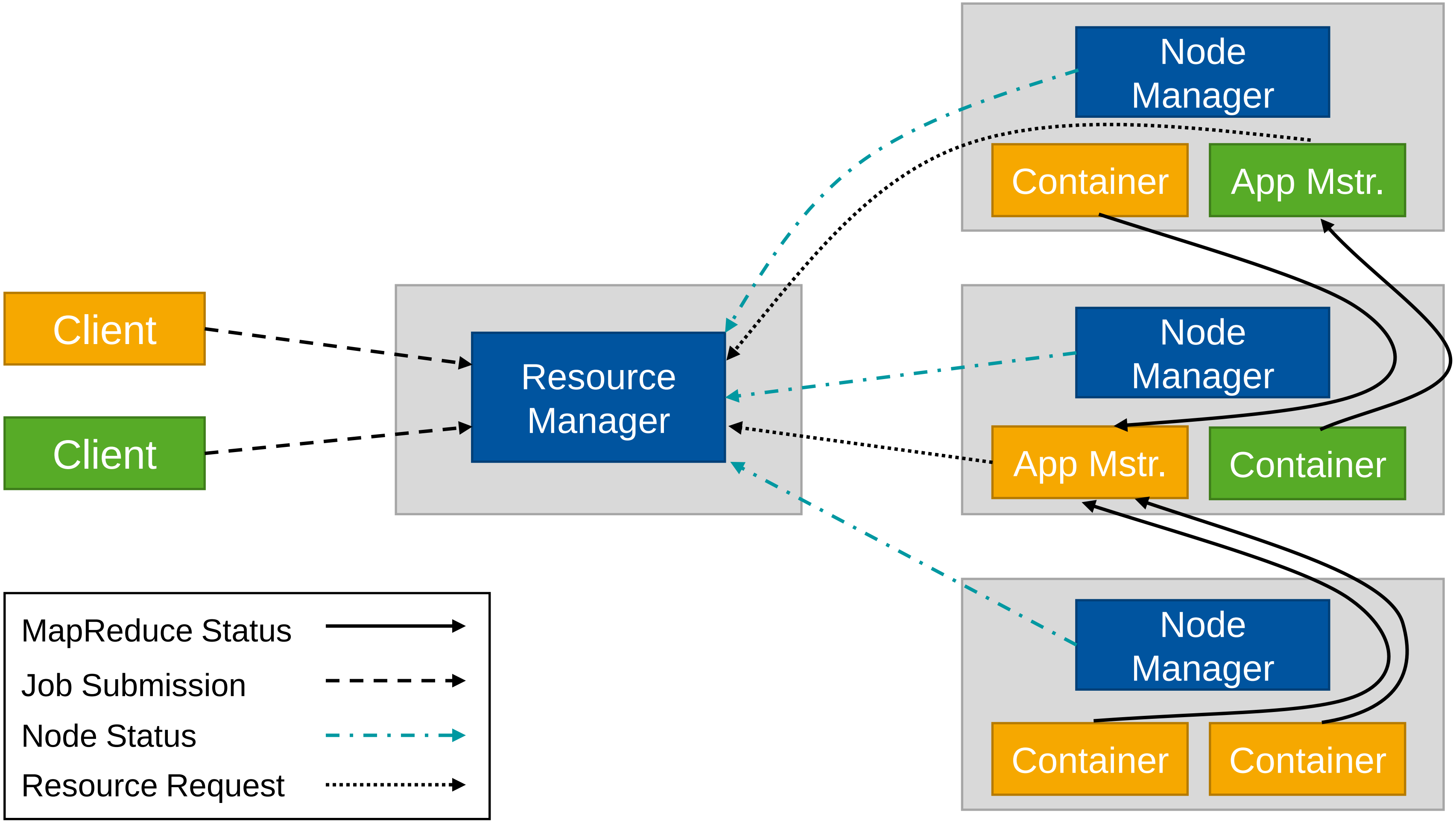


Illustration adapted from https://hadoop.apache.org/docs/r3.1.0/hadoop-yarn/hadoop-yarn-site/YARN.html

YARN and HDFS

- Both YARN and HDFS: Master / Worker architecture
- Typical cluster setup: Two machines as masters and rest workers
  - HDFS NameNode on one machine (master)
  - ResourceManager on another machine (master)
  - Other machines: HDFS DataNode and NodeManager simultaneously (workers)
- Storage handling done by HDFS
- Computation handling done by YARN

Task Assignments – Data Locality (1)

- Performance bottleneck for data-intensive task: Network bandwidth
- Data management concept of HDFS: **Data Locality**
- Container request in YARN can have locality constraints
  - Request to run task on a certain node
- Application master knows from NameNode which data lies on which node
  - Include locality constraint in container request
- Map task: Processing HDFS blocks locally
  - Ideally: Container runs on a node storing a replica of accessed HDFS blocks

Task Assignments – Data Locality (2)

- Locality levels
  - Data-local (optimal, task runs on same node which stores needed data)
  - Rack-local (same rack, but not same node)
- Data locality only possible for map tasks
  - Reduce tasks collect data from different machines, no data locality
- **Goal:** Process most input data (of map tasks) locally → Less network bandwidth needed

Task Granularity

- Map phase: *M* map tasks
- Reduce phase: *R* reduce tasks
- Ideally: *M* and *R* much larger than the number of worker nodes
  - Better load balancing
  - Faster recovery in case of a failure
- Too large values for *M* and *R* can lead to significant overhead at ResourceManager

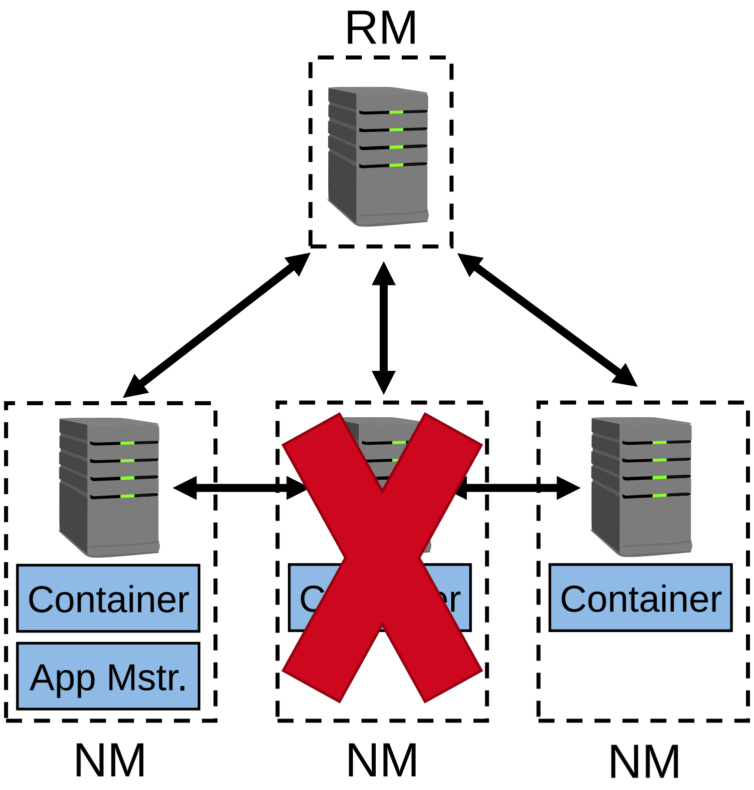
- In practice: $M = 200{,}000$ and $\mathrm{R} = 5{,}000$ with $2{,}000$ worker machines, each map task processes about 16 MB to 64 MB (Google, 2008)

Fault Tolerance

- MapReduce programs run on large number of machines
- Implementation must tolerate and handle process crashes, machine failures, …
- Reasons for failures
  - Bad user codes, hardware faults, network connection issues, OS crash, …
- Four failure types
  - Task failure: Reschedule task or abort job execution
  - Application master failure: Start new application master on other NM, job recovery required
  - **NodeManager failure**
  - **ResourceManager failure**

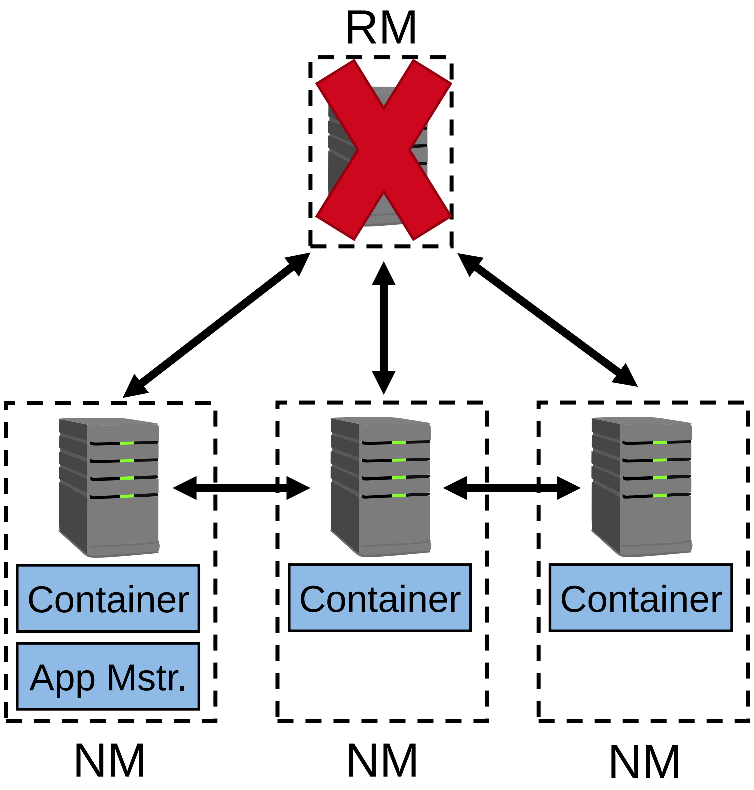
Fault Tolerance – Node Manager Failure

- Periodic heartbeats sent from NM to RM
- On NM failure: Absence of heartbeats
- RM detects problem, potentially recovers corresponding tasks and application master
- Additionally: Completed map tasks have to be rerun for incomplete jobs
  - Intermediate output on the failed NM's local file system might be inaccessible



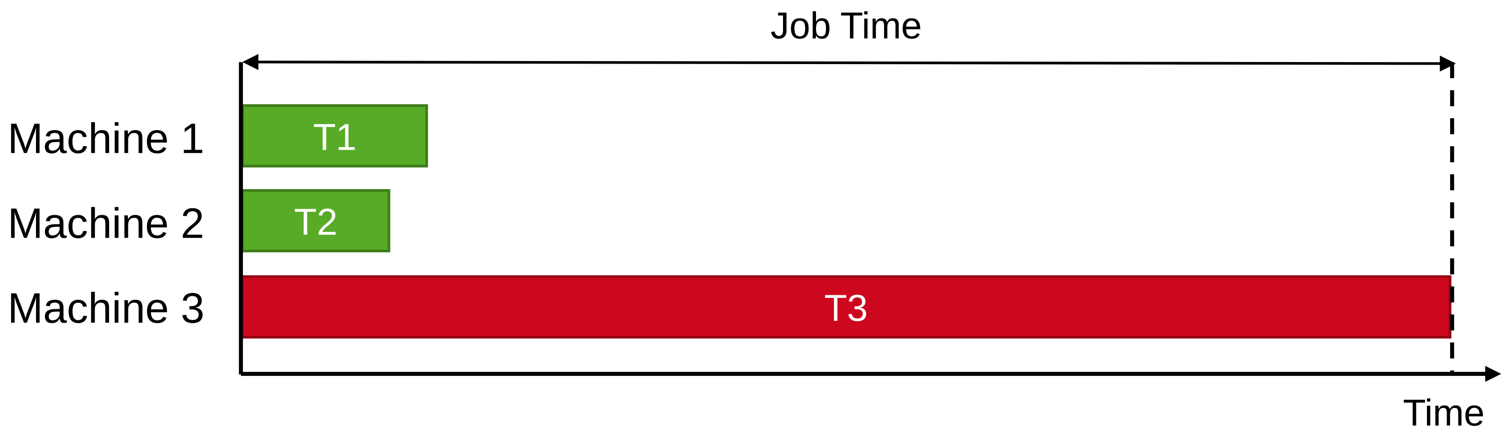
Fault Tolerance – Resource Manager Failure

- Resource manager is single point of failure (SPOF)
  - No jobs or task containers can be started without it
- Similar to HDFS NameNode: Run pair of resource managers to resolve SPOF
  - Active-standby configuration
  - State is stored in shared storage
- On failure of active resource manager
  - Standby resource manager can recover state from storage and take over



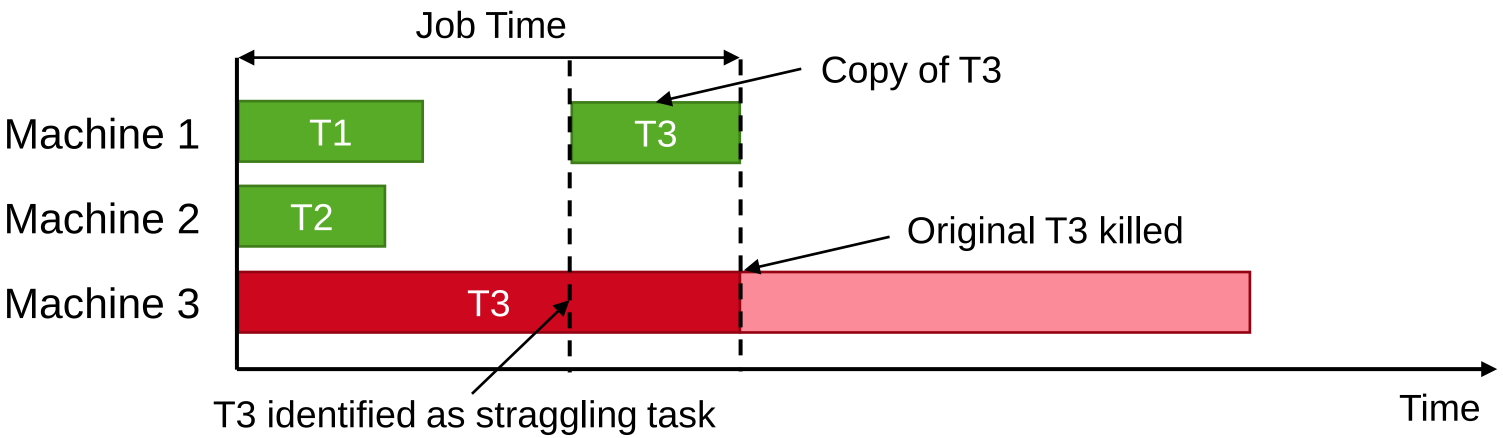
Speculative Execution (1)

- Critical for total execution time: Straggler
  - Machine taking unusually long time to complete one of the last map or reduce tasks
- Reasons for stragglers
  - Bad disk with slow read performance
  - Other tasks on the same machine
  - Bugs in machine configuration
- Probability of a straggler high for thousands of machines

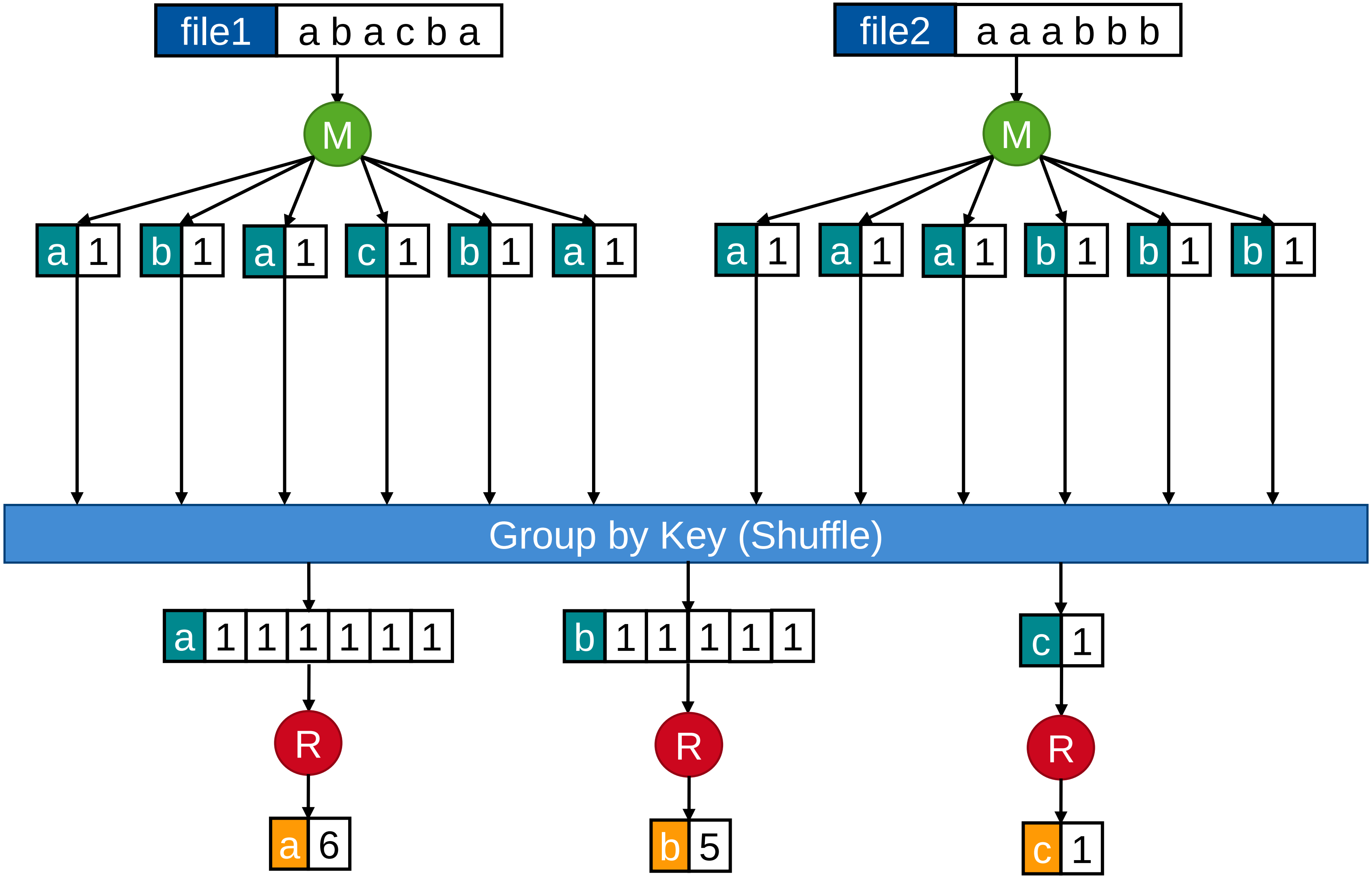


Speculative Execution (2)

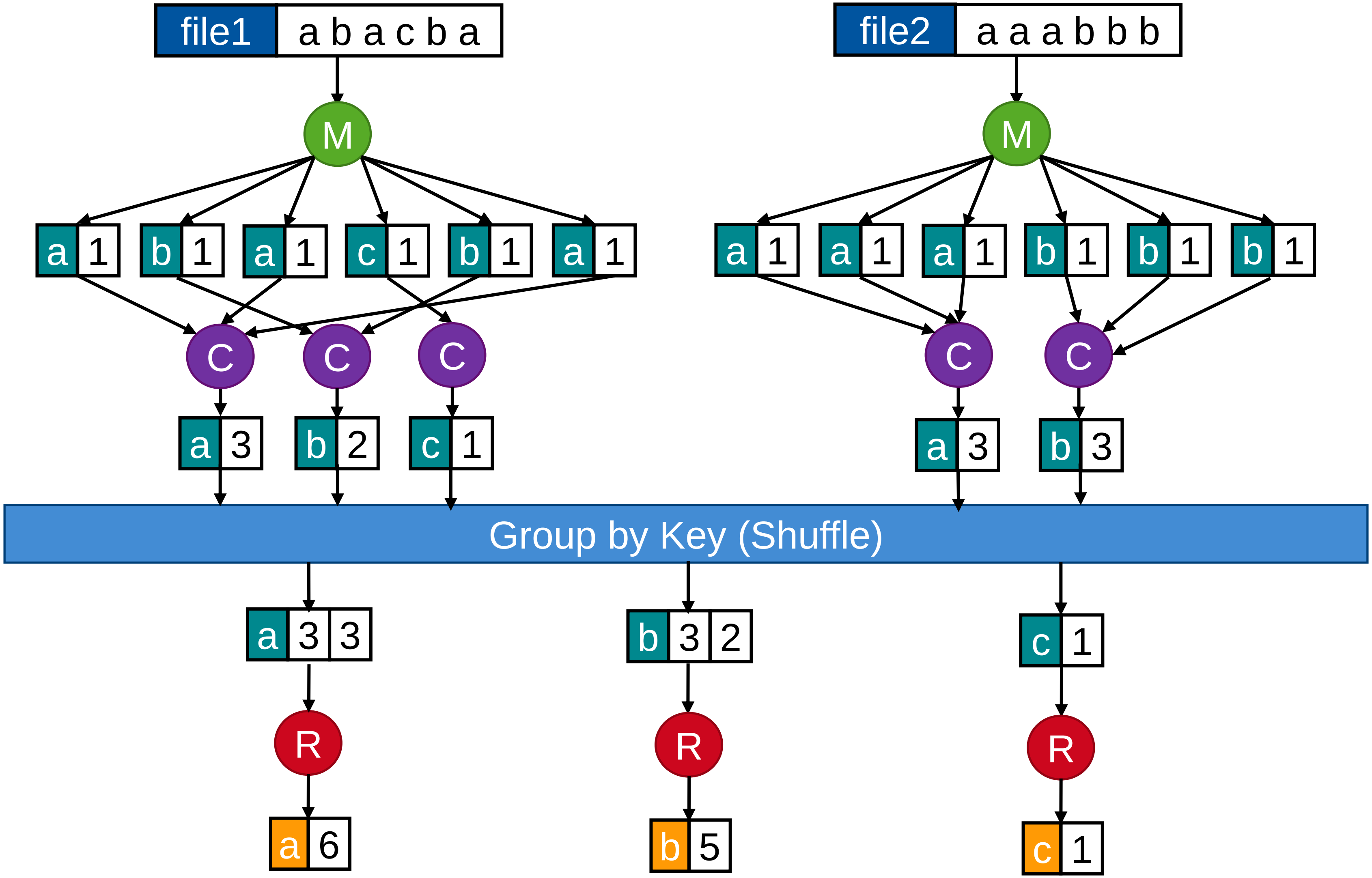
- Solution: Speculative execution
  - Detect tasks running at least one minute and with much less progress than the other tasks (on average) → Straggling task
  - If straggling task detected: Launch duplicate task as backup
  - Task is completed if primary or backup task completes, other instance is killed
- In practice: Significant reduction of total time
  - ≈ 30 % time reduction for a sorting program with 1,800 machines (Google)
- Drawback: Additional load on the cluster



Word Count Example – Revisited



Combiner Function – Word Count



$$Map(k_1, v_1) \rightarrow list(k_2, v_2)$$
$$Reduce\big(k_2, list(v_2)\big) \rightarrow list(k_3, v_3)$$

Combiner Function

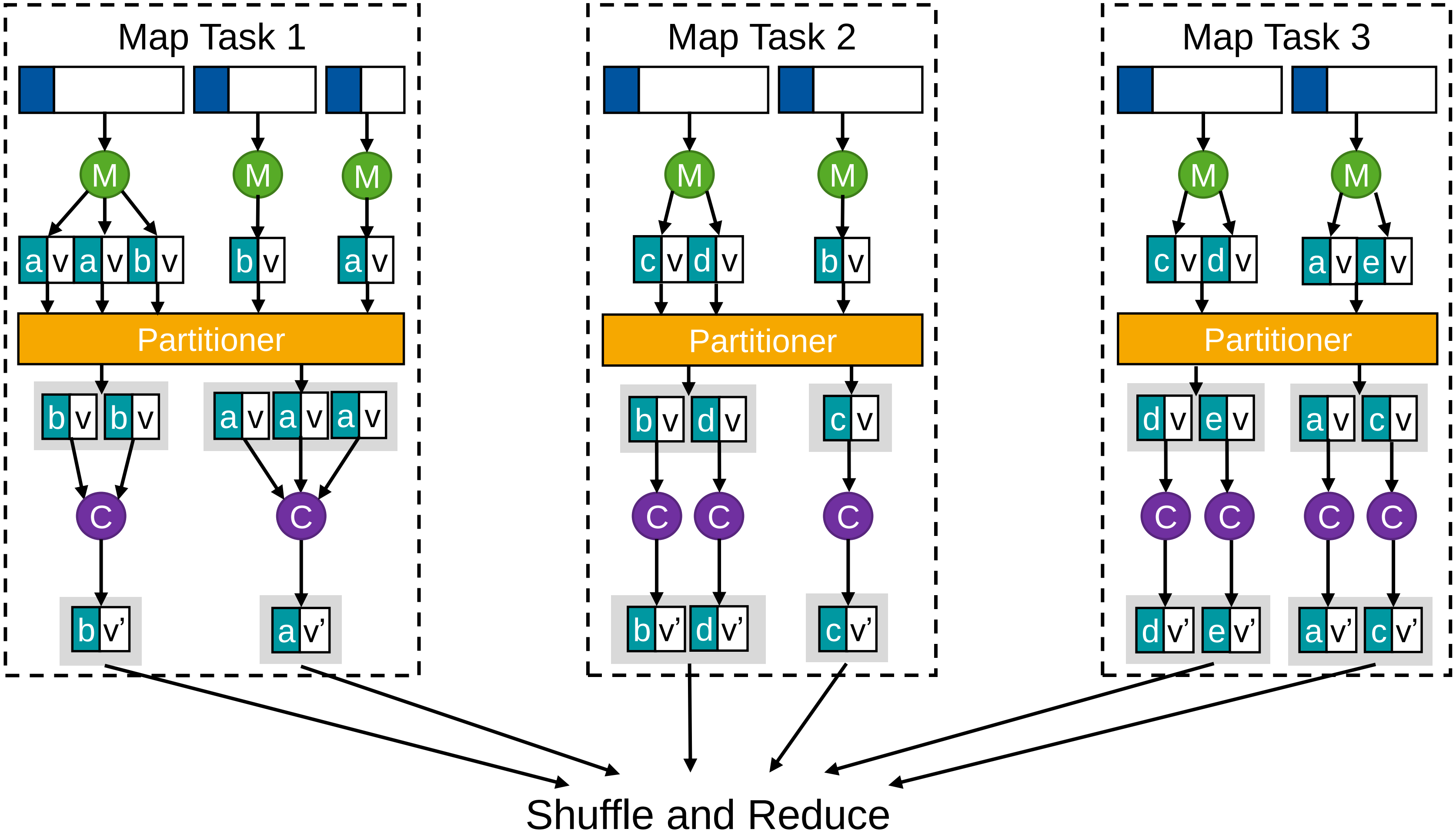
- Limit amount of data transferred between map and reduce task
- Idea: Aggregate (if possible) map output before reduction (“mini-reduce”)
- Optimization of Hadoop: Combiner function

$$Combine(k_2, list(v_2)) \rightarrow list(k_2, v_2)$$

  - Runs on partitioned map output and aggregates (“combines”) list of items to a smaller list
  - Runs directly after partitioning
  - Potentially performed for each map task
- No guarantee how often it is called for a particular map task (zero or more invocations)
  - Computation should be independent of combiner invocations
  - The combiner is <u>not</u> allowed to change the key of the input KV pair, it can only aggregate the list of values.
- Word count example: Aggregate word counts for each map output

Illustration adapted from Dean, Jeffrey, and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters.” *Communications of the ACM* 51.1 (2008): 107-113.

MapReduce in Parallel with Combiner



Job Submission with Combiner in Hadoop

Job configuration and submission:

```
public class WordCount {
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);

    job.setMapperClass(WordCountMapper.class);
    job.setCombinerClass(WordCountReducer.class);
    job.setReducerClass(WordCountReducer.class);

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```