Distributed Computations MapReduce

adapted from Jeff Dean's slides

What we've learnt so far

- Basic distributed systems concepts
  - Consistency (sequential, eventual)
  - Fault tolerance (recoverability, availability)
- What are distributed systems good for?
  - Better fault tolerance
    - Better security?
  - Increased storage/serving capacity
    - Storage systems, email clusters
  - Parallel (distributed) computation (Today's topic)

Why distributed computations?

- How long to sort 1 TB on one computer?
  – One computer can read ~30MB from disk
  – Takes ~2 days!!
- Google indexes 20 billion+ web pages
  – 20 * 10^9 pages * 20KB/page = 400 TB
- Large Hadron Collider is expected to produce 15 PB every year!

Solution: use many nodes!

- Cluster computing
  - Hundreds or thousands of PCs connected by high speed LANs
- Grid computing
  - Hundreds of supercomputers connected by high speed net
- 1000 nodes potentially give 1000X speedup

Distributed computations are difficult to program

- Sending data to/from nodes
- Coordinating among nodes
- Recovering from node failure
- Optimizing for locality
- Debugging

Same for all problems

MapReduce

- A programming model for large-scale computations
  - Process large amounts of input, produce output
  - No side-effects or persistent state (unlike file system)

- MapReduce is implemented as a runtime library:
  - automatic parallelization
  - load balancing
  - locality optimization
  - handling of machine failures

MapReduce design

- Input data is partitioned into M splits
- Map: extract information on each split
  - Each Map produces R partitions
- Shuffle and sort
  - Bring M partitions to the same reducer
- Reduce: aggregate, summarize, filter or transform
- Output is in R result files

More specifically…

- Programmer specifies two methods:
  - map(k, v) → <k', v'>*
  - reduce(k', <v'>*) → <k', v'>*
- All v' with same k' are reduced together, in order.

- Usually also specify:
  - partition(k’, total partitions) -> partition for k’
    - often a simple hash of the key
    - allows reduce operations for different k’ to be parallelized

Example: Count word frequencies in web pages

- Input is files with one doc per record
- Map parses documents into words
  - key = document URL
  - value = document contents
- Output of map:

“doc1”, “to be or not to be” →

“to”, “1”
“be”, “1”
“or”, “1”
…

Example: word frequencies

- Reduce: computes sum for a key

key = “be”
values = “1”, “1”

“2”

key = “not”
values = “1”

“1”

key = “or”
values = “1”

“1”

key = “to”
values = “1”, “1”

“2”

- Output of reduce saved

“be”, “2”
“not”, “1”
“or”, “1”
“to”, “2”

Example: Pseudo-code

```
Map(String input_key, String input_value):
  //input_key: document name
  //input_value: document contents
  for each word w in input_values:
    EmitIntermediate(w, "1");

Reduce(String key, Iterator
intermediate_values):
  //key: a word, same for input and output
  //intermediate_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v);
  Emit(AsString(result));
```

MapReduce is widely applicable

- Distributed grep
- Document clustering
- Web link graph reversal
- Detecting approx. duplicate web pages
- …

MapReduce implementation

- Input data is partitioned into M splits
- Map: extract information on each split
  - Each Map produces R partitions
- Shuffle and sort
  - Bring M partitions to the same reducer
- Reduce: aggregate, summarize, filter or transform
- Output is in R result files

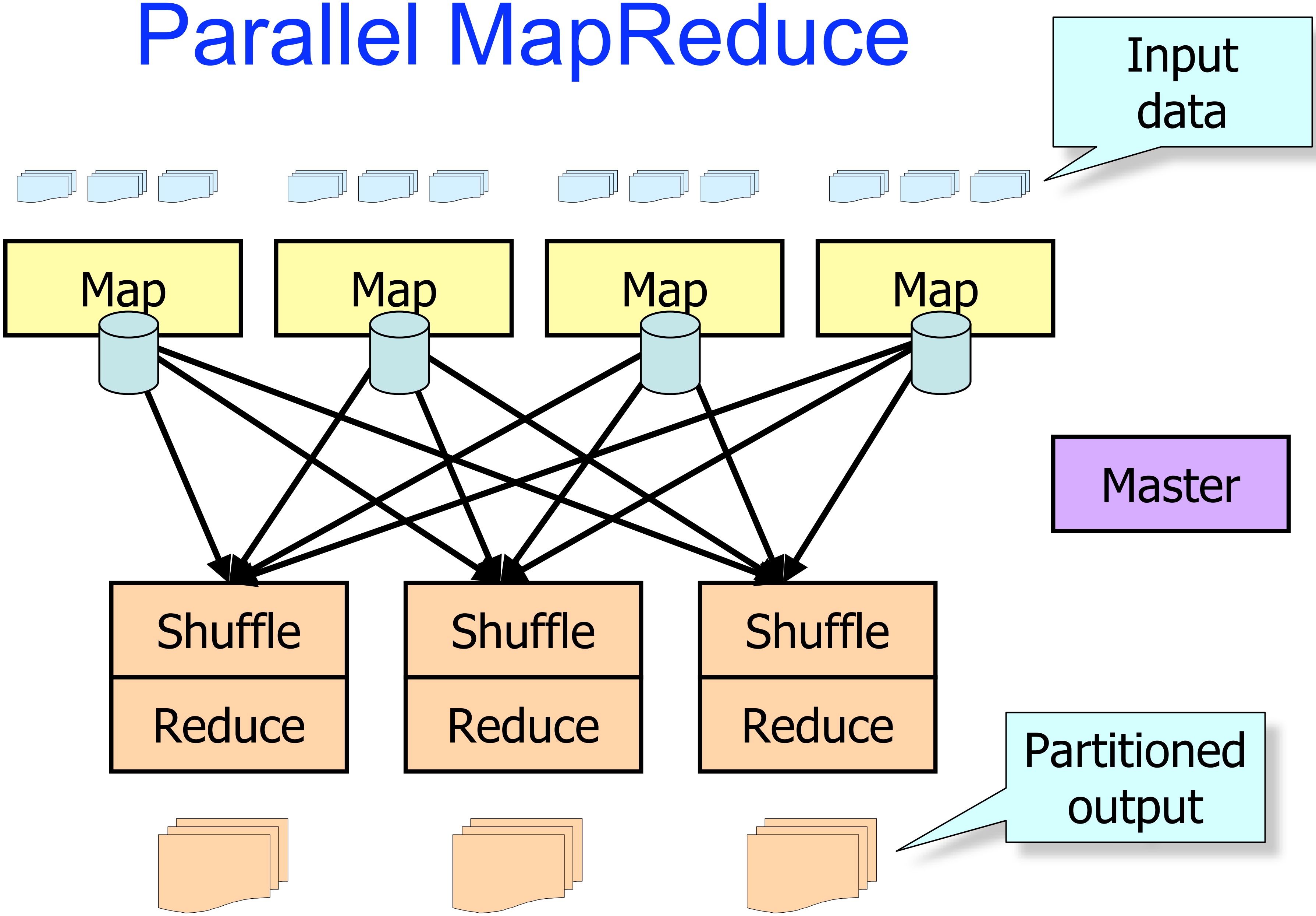
MapReduce scheduling

- One master, many workers
  - Input data split into *M* map tasks (e.g. 64 MB)
  - *R* reduce tasks
  - Tasks are assigned to workers dynamically
  - Often: *M*=200,000; *R*=4,000; workers=2,000

MapReduce scheduling

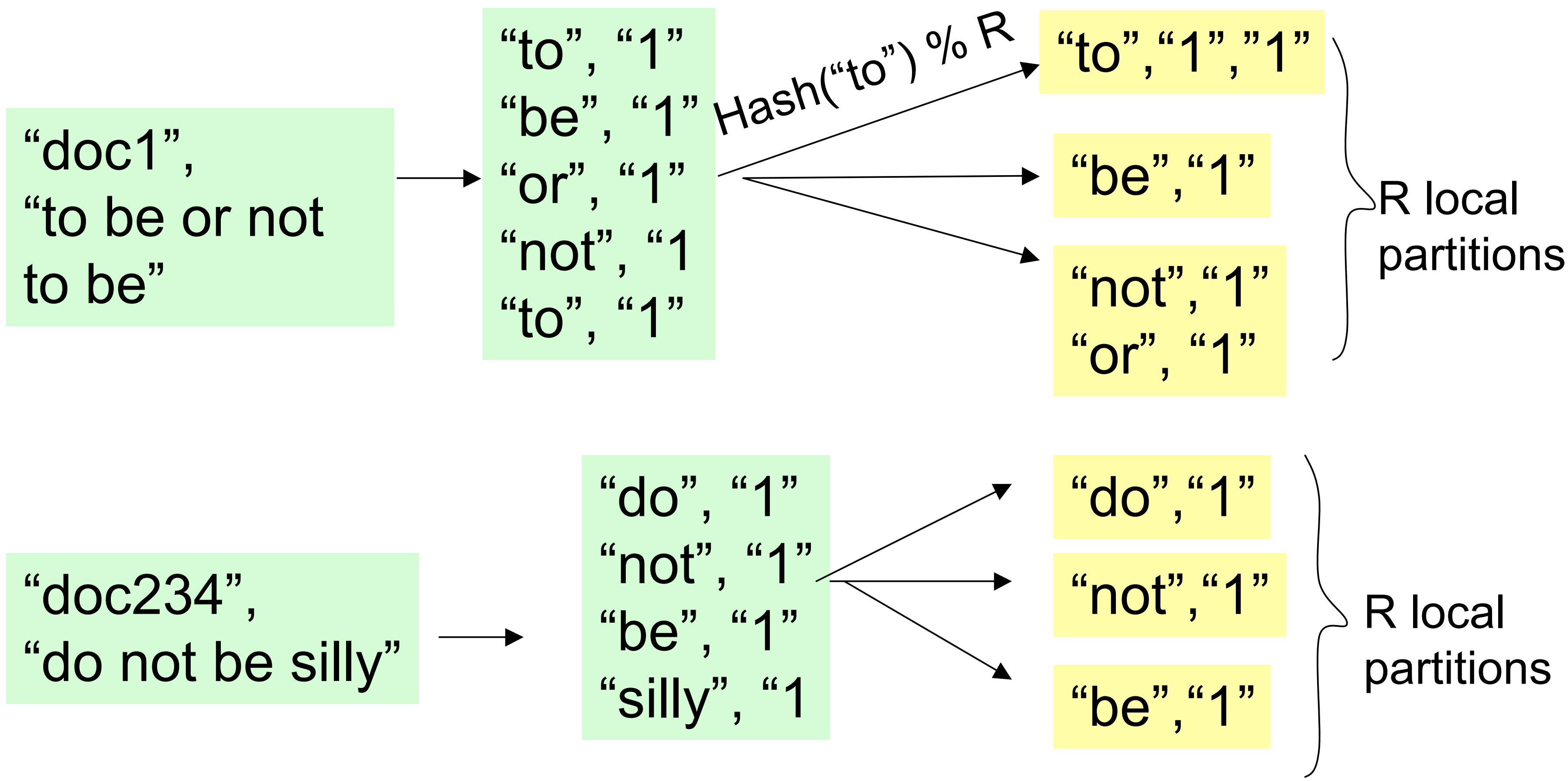
- Master assigns a map task to a free worker
  - Prefers “close-by” workers when assigning task
  - Worker reads task input (often from local disk!)
  - Worker produces R **local files** containing intermediate k/v pairs
- Master assigns a reduce task to a free worker
  - Worker reads intermediate k/v pairs from map workers
  - Worker sorts & applies user’s *Reduce* op to produce the output

Parallel MapReduce



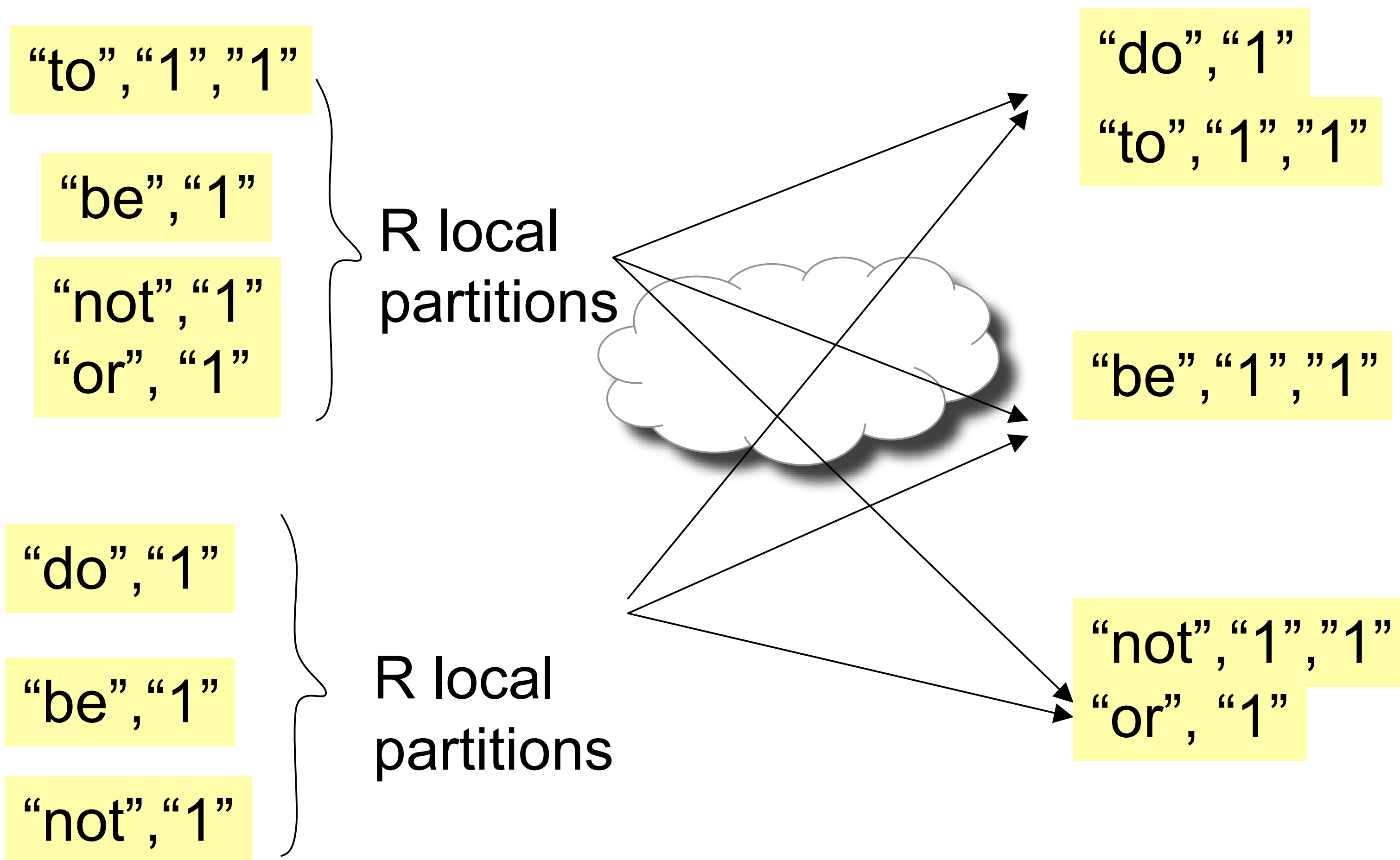
WordCount Internals

- Input data is split into M map jobs
- Each map job generates in R local partitions



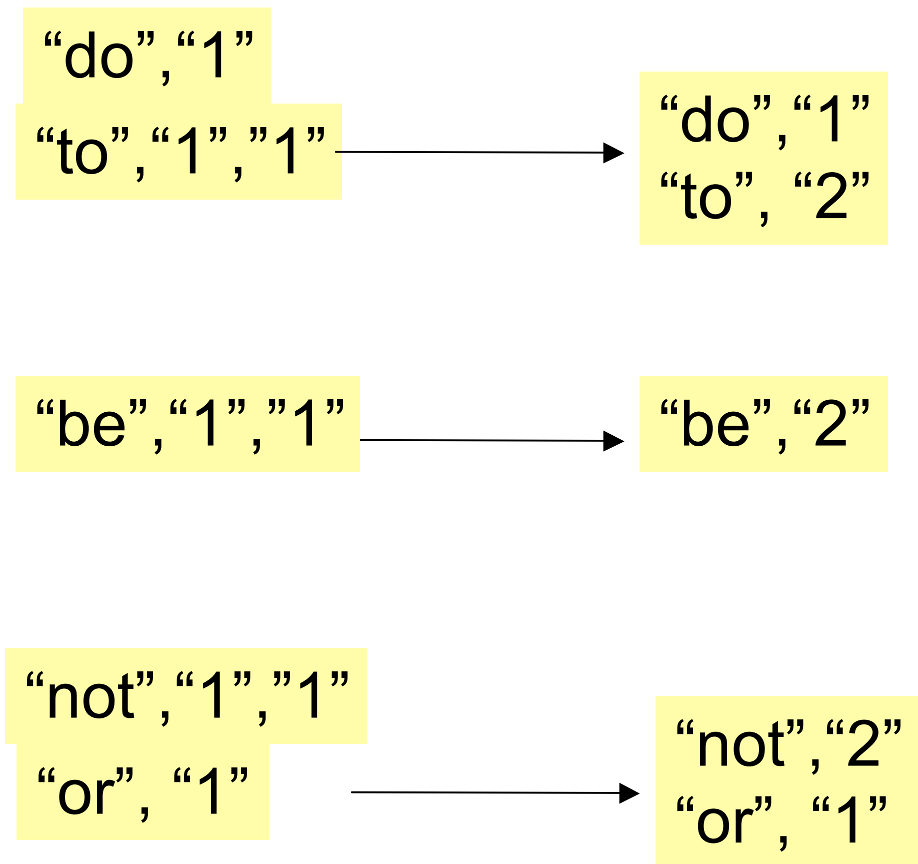
WordCount Internals

- Shuffle brings same partitions to same reducer



WordCount Internals

- Reduce aggregates sorted key values pairs

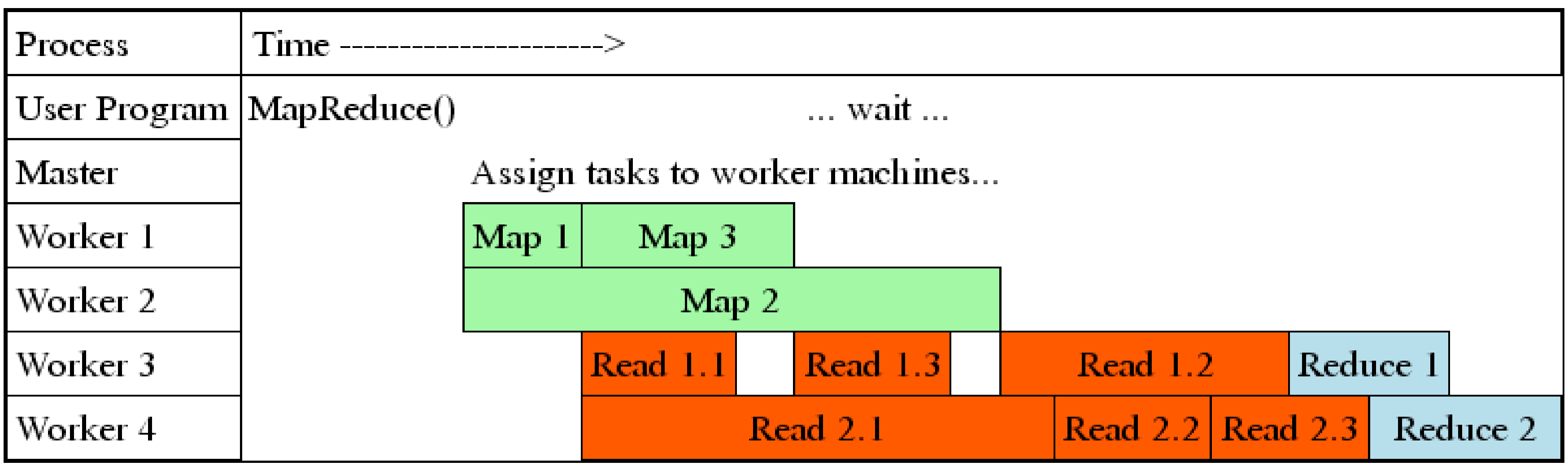


The importance of partition function

- partition(k’, total partitions) -> partition for k’
  - e.g. hash(k’) % R
- What is the partition function for sort?

Load Balance and Pipelining

- Fine granularity tasks: many more map tasks than machines
  - Minimizes time for fault recovery
  - Can pipeline shuffling with map execution



Fault tolerance via re-execution

On worker failure:

- Re-execute completed and in-progress map tasks
- Re-execute in progress reduce tasks
- Task completion committed through master

On master failure:

- State is checkpointed to GFS: new master recovers & continues

Avoid straggler using backup tasks

- Slow workers significantly lengthen completion time
  - Other jobs consuming resources on machine
  - Bad disks with soft errors transfer data very slowly
  - Weird things: processor caches disabled (!!)
  - An unusually large reduce partition?
- Solution: Near end of phase, spawn backup copies of tasks
  - Whichever one finishes first "wins"
- Effect: Dramatically shortens job completion time

MapReduce Sort Performance

- 1TB (100-byte record) data to be sorted
- 1700 machines
- M=15000 R=4000

MapReduce Sort Performance