Lecture 17: MapReduce

Principles of Computer Systems

Autumn 2019

Stanford University

Computer Science Department

Instructors: Chris Gregg

Philip Levis

PDF of this presentation

The Genesis of Datacenter Computing: MapReduce

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined

The Genesis of Datacenter Computing: MapReduce

- Problem for large service providers such as Google: computation requires hundreds or thousands of computers
  - How do you distribute the computation?
  - Distributing the computation boils down to distributing data
  - Nodes fail, some nodes are slow, load balancing: difficult to make it work well
- System came from observation that many early Google systems had a common pattern
- Designing a computing system around this pattern
  - allows the system (written once) to do all of the hard systems work (fault tolerance, load balancing, parallelization)
  - allows tens of thousands of programmers to just write their computation
- Canonical example of how the right *abstraction* revolutionized computing
  - An open source version immediately appeared: Hadoop

Core Data Abstraction: Key/Value Pairs

- Take a huge amount of data (more than can fit in the memory of 1000 machines)
- Write two functions:

```
map(k1, v1) -> list(k2, v2)
reduce(k2, list(v2)) -> list(v2)
```

- Using these two functions, MapReduce parallelizes the computation across thousands of machines, automatically load balancing, recovering from failures, and producing the correct result.
- You can string together MapReduce programs: output of reduce becomes input to map.
- Simple example of word count (wc):

```
1  map(String key, String value):
2    // key: document name
3    // value: document contents
4    for word w in value:
5      EmitIntermediate(w,"1")
6
7  reduce(String key, List values):
8    // key: a word
9    // values: a list of counts
10   int result = 0
11   for v in values:
12     result += ParseInt(v)
13   Emit(AsString(result))
```

input	"The number of partitions (R) and the partitioning function are specified by the user."
map output	("the", "1") , ("the", "1"), ("The", "1"), ("of", 1), (number, "1"), ...
reduce	"the", ("1", "1") -> "2" "The", ("1") -> "1" "of", ("1") -> "1" "number", ("1") -> "1"

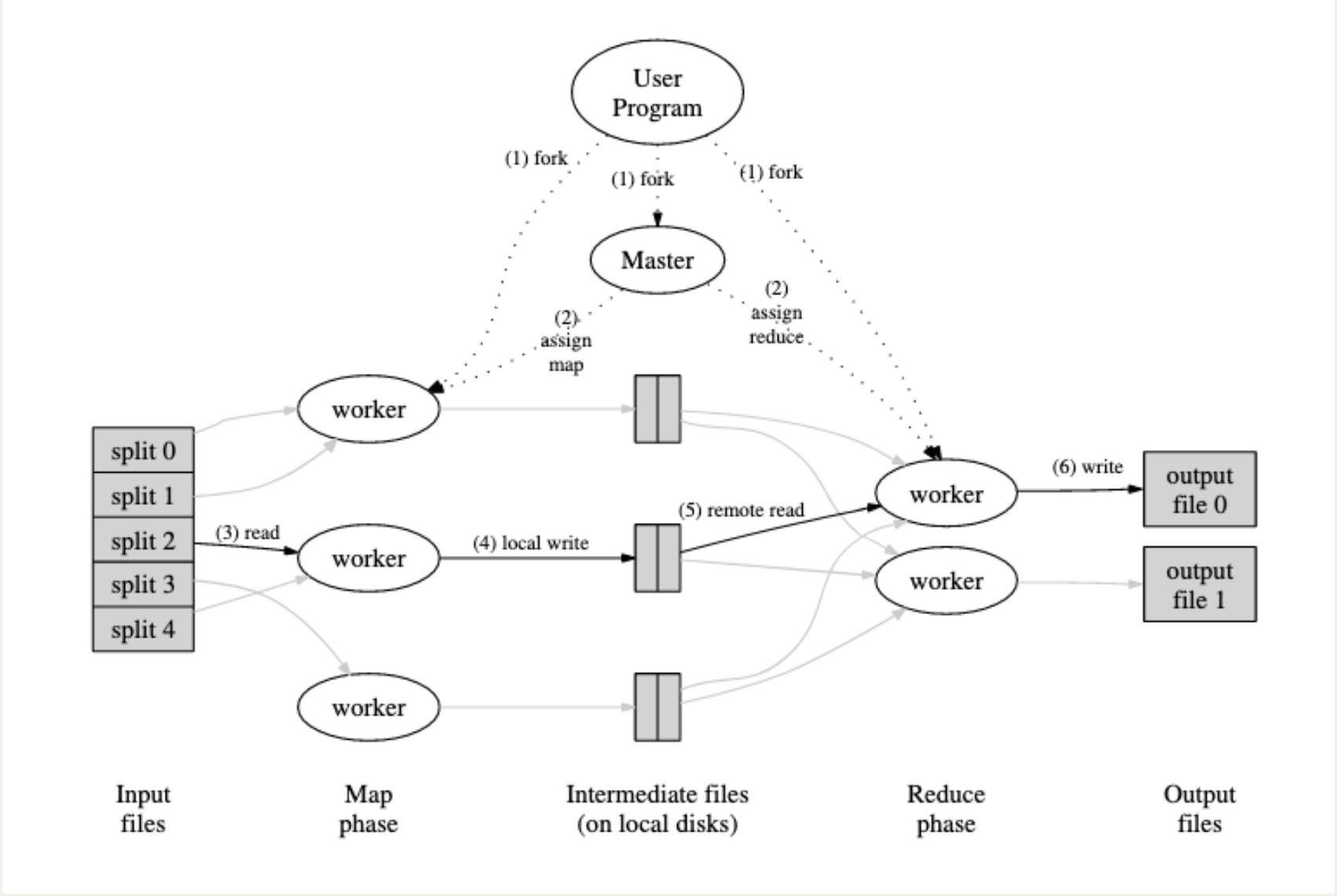
Key/Value Pairs: How and Where

- Keys allow MapReduce to distribute and parallelize load

```
map(k1, v1) -> list(k2, v2)
reduce(k2, list(v2)) -> list(v2)
```

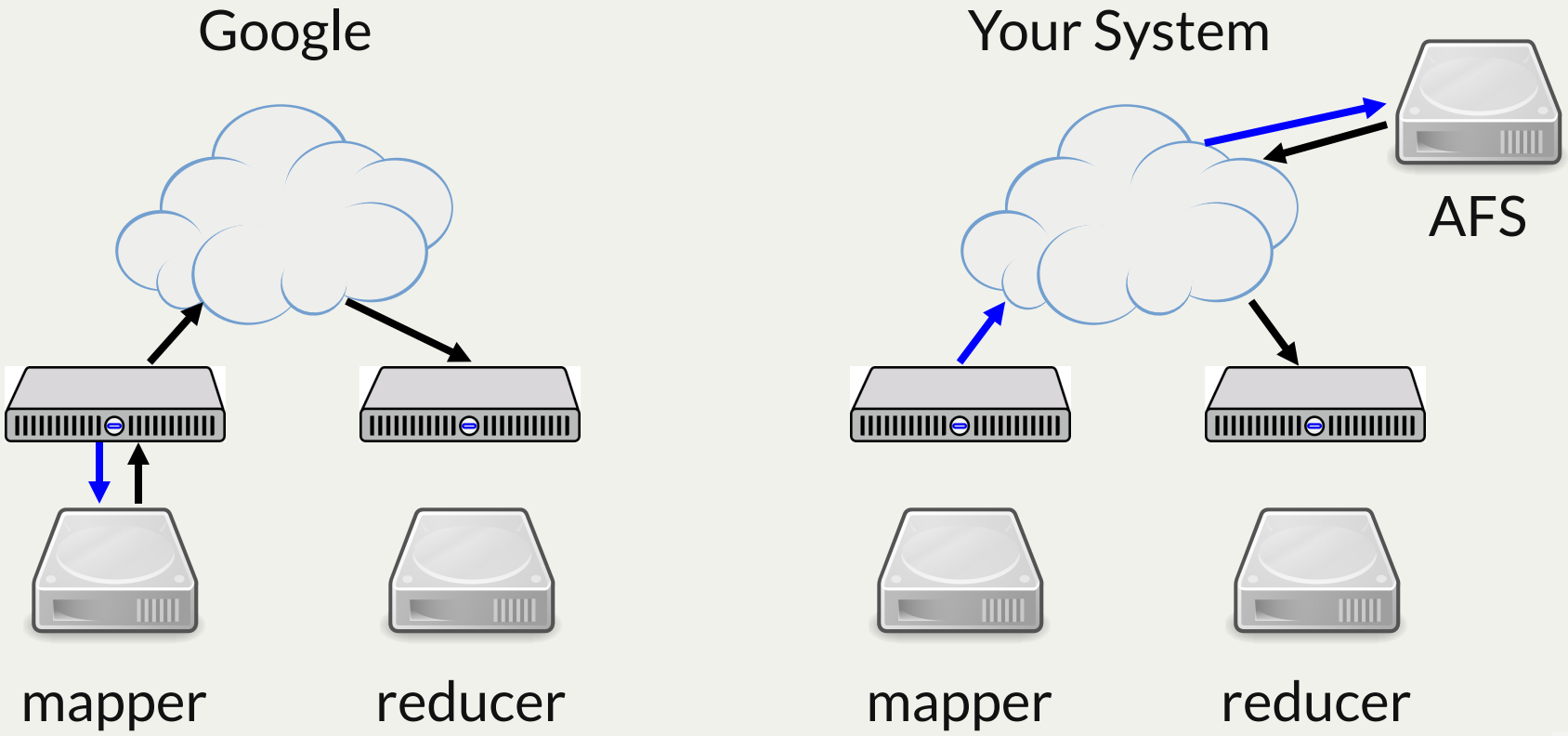
- Core abstraction: data can be partitioned by key, there is no locality between keys
- In the original paper...
  - Each *mapper* writes a local file for each key in k2, and reports its files to a *master* node
  - The *master* node tells the *reducer* for k2 where the all of the k2 files are
  - The reducer reads all of the k2 files from the nodes that ran the mappers and writes its own output locally, reporting this to the master node
  - There have been lots of optimizations since
- Keys can be arbitrary data: hash the keys, and assign keys to splits using modulus
  - split(key) = hash(key) % N, where N is the number of splits
- A master node tracks progress of workers and manages execution
  - When a worker falls idle, the master sends it a new split to compute
  - When the master thinks a worker has failed, tell another worker to compute its split

MapReduce System Architecture



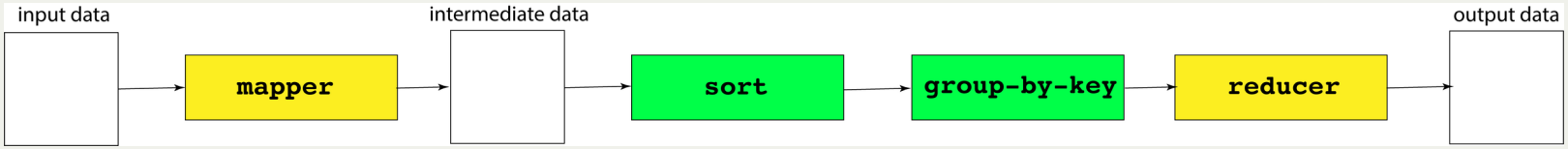
Your MapReduce

- Your MapReduce will differ from the standard one in one significant way: instead of writing results to local disk, it writes results to AFS, myth's networked file system
  - Every worker can access the files directly using AFS
  - Cost: file write is over the network
  - Benefit: recovery from failure is easier (don't have to regenerate lots files)
  - Benefit: don't have to write file transfer protocol (handled by AFS)



MapReduce Data Flow

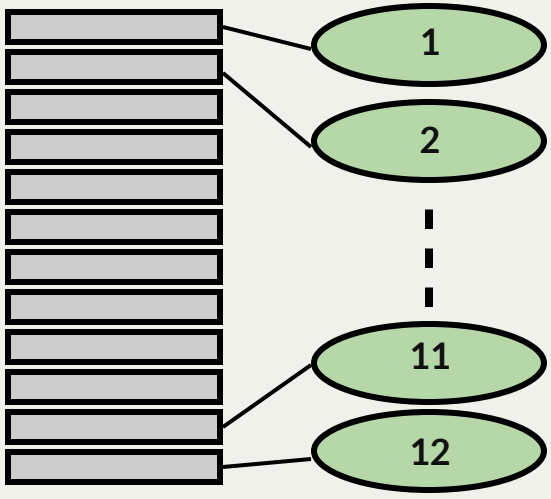
- The **map** component of a MapReduce job typically parses input data and distills it down to some intermediate result.
- The **reduce** component of a MapReduce job collates these intermediate results and distills them down even further to the desired output.
- The pipeline of processes involved in a MapReduce job is captured by the below illustration:



- The processes shaded in yellow are programs specific to the data set being processed, whereas the processes shaded in green are present in all MapReduce pipelines.
- We'll invest some energy over the next several slides explaining what a mapper, a reducer, and the group-by-key processes look like.

Word Count Example

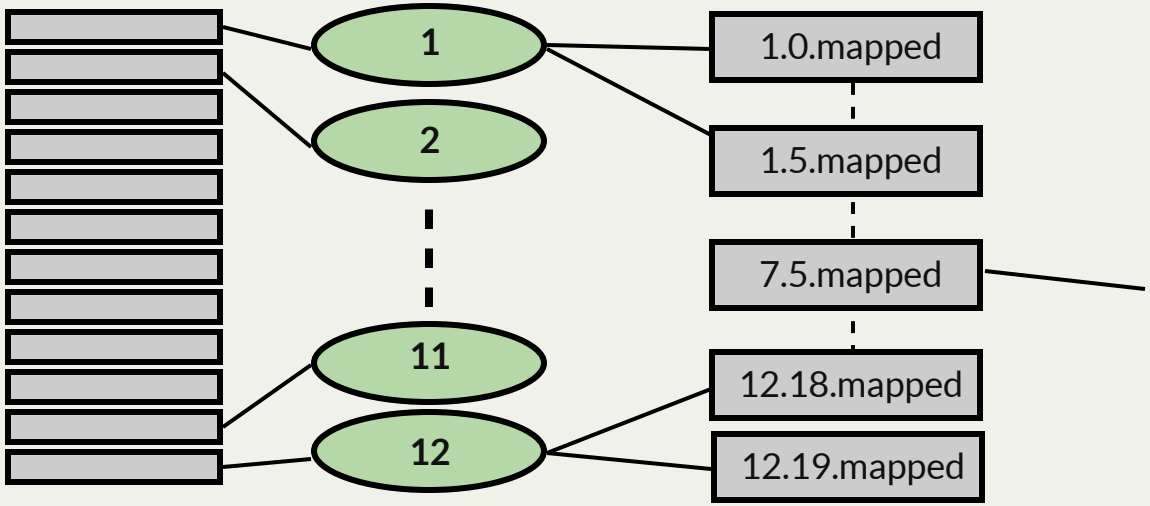
- Walk through word count example in detail, see what MapReduce does
- There are a bunch of parameters, let's set them so
  - Number of map tasks (input partitions/splits): 12
    - In normal MapReduce this is user-specifiable, in your implementation this is predefined by how the input is split
  - Number of map workers: 4
  - Number of reduce tasks (intermediate and output partitions/splits): 20
  - Number of reduce workers: 5
- Each map task (assigned to one of 4 map workers) maps an input partition




```
1  map(String key, String value):
2    // key: document name
3    // value: document contents
4    for word w in value:
5      EmitIntermediate(w,"1")
6
7  reduce(String key, List values):
8    // key: a word
9    // values: a list of counts
10   int result = 0
11   for v in values:
12     result += ParseInt(v)
13   Emit(AsString(result))
```

Word Count Example

- Walk through word count example in detail, see what MapReduce does
- There are a bunch of parameters, let's set them so
  - Number of map tasks (input partitions/splits): 12
    - In normal MapReduce this is user-specifiable, in your implementation this is predefined by how the input is split
  - Number of map workers: 4
  - Number of reduce tasks (intermediate and output partitions/splits): 20
  - Number of reduce workers: 5
- Each map task produces 20 output files; each file F contains hash(key) % 20 == F



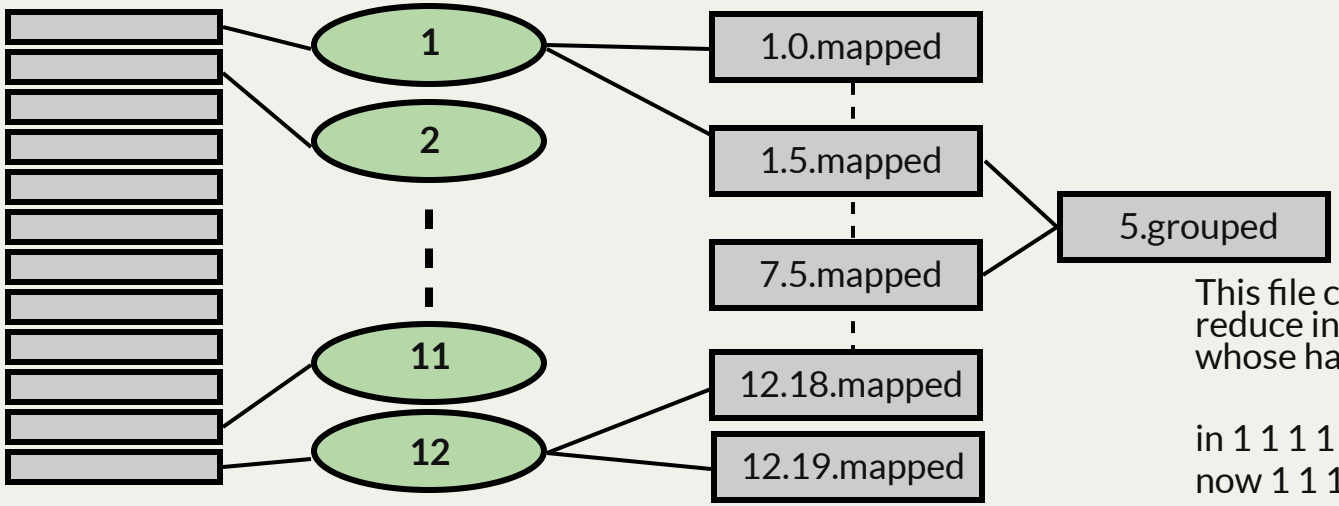
This file contains all of the map outputs for map task 7 whose key, modulo 20, is 5

in 1
now 1
step 1
now 1
in 1
in 1
....

"Propitious Pallas, to secure..."

Word Count Example

- Walk through word count example in detail, see what MapReduce does
- There are a bunch of parameters, let's set them so
  - Number of map tasks (input partitions/splits): 12
    - In normal MapReduce this is user-specifiable, in your implementation this is predefined by how the input is split
  - Number of map workers: 4
  - Number of reduce tasks (intermediate and output partitions/splits): 20
  - Number of reduce workers: 5
- Runtime (your reducer) collates, sorts, and groups all of the inputs for each reduce task



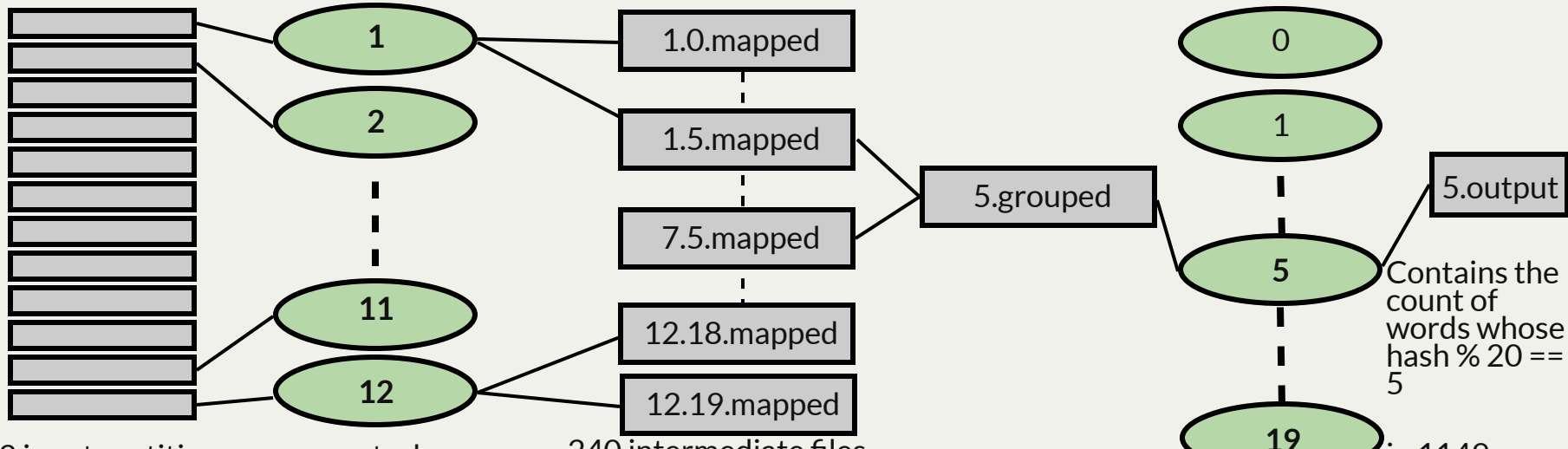
This file contains all of the reduce inputs for every key whose hash % 20 == 5

in 1 1 1 1 1 1 1 1 .. *(1149 times)*
now 1 1 1 1 ... (126 times)
step 1 1 1 1 ... (10 times)
....

"Propitious Pallas, to secure..."

Word Count Example

- Walk through word count example in detail, see what MapReduce does
- There are a bunch of parameters, let's set them so
  - Number of map tasks (input partitions/splits): 12
    - In normal MapReduce this is user-specifiable, in your implementation this is predefined by how the input is split
  - Number of map workers: 4
  - Number of reduce tasks (intermediate and output partitions/splits): 20
  - Number of reduce workers: 5
- Each reduce task (scheduled to one of 5 workers) runs reduce on its input



"Propitious Pallas, to secure..."

Importance of Keys

- Keys are the mechanism that allows MapReduce to distribute load across many machines while keeping data locality
  - All data with the same key is processed by the same mapper or reducer
  - Any map worker or reduce worker can process a given key
  - Keys can be collected together into larger units of work by hashing
    - We've seen this: if there are N tasks, a key K is the responsibility of the task whose ID is hash(K) % N
- Distinction between tasks and nodes: there are many more tasks than nodes
  - Worker nodes request work from the master server, which sends them map or reduce tasks to execute
  - If one task is fast, the worker just requests a new one to complete
  - Very simple load balancing
  - This load balancing works because a task can run anywhere

MapReduce Assignment: Example

- Let's see an example run with the solution executables:

```
myth57:$ make filefree
rm -fr files/intermediate/* files/output/*

myth57:$ ./samples/mr_soln --mapper ./samples/mrm_soln --reducer ./samples/mrr_soln --config odyssey-full.cfg

Determining which machines in the myth cluster can be used... [done!!]
Mapper executable: word-count-mapper
Reducer executable: word-count-reducer
Number of Mapping Workers: 8
Number of Reducing Workers: 4
Input Path: /usr/class/cs110/samples/assign8/odyssey-full
Intermediate Path: /afs/.ir.stanford.edu/users/c/g/cgregg/cs110/spring-2019/assignments/assign8/files/intermediate
Output Path: /afs/.ir.stanford.edu/users/c/g/cgregg/cs110/spring-2019/assignments/assign8/files/output
Server running on port 48721

Received a connection request from myth59.stanford.edu.
Incoming communication from myth59.stanford.edu on descriptor 6.
Instructing worker at myth59.stanford.edu to process this pattern: "/usr/class/cs110/samples/assign8/odyssey-full/00001.input"
Conversation with myth59.stanford.edu complete.
Received a connection request from myth61.stanford.edu.
Incoming communication from myth61.stanford.edu on descriptor 7.

... LOTS of lines removed

Remote ssh command on myth56 executed and exited with status code 0.
Reduction of all intermediate chunks now complete.
/afs/.ir.stanford.edu/users/c/g/cgregg/cs110/spring-2019/assignments/assign8/files/output/00000.output hashes to 13585898109251157014
/afs/.ir.stanford.edu/users/c/g/cgregg/cs110/spring-2019/assignments/assign8/files/output/00001.output hashes to 1022930401727915107
/afs/.ir.stanford.edu/users/c/g/cgregg/cs110/spring-2019/assignments/assign8/files/output/00002.output hashes to 9942936493001557706
/afs/.ir.stanford.edu/users/c/g/cgregg/cs110/spring-2019/assignments/assign8/files/output/00003.output hashes to 5127170323801202206

... more lines removed
```

- There is a plethora of communication between the machine we run on and the other myths.
- Output ends up in the **`files/`** directory.

MapReduce Assignment: Mapped File Contents

- The map phase of **`mr`** has the 8 mappers (from the .cfg file) process the 12 files processed by **`word-count-mapper`** and put the results into **`files/intermediate`**:

```
myth57:$ ls -lu files/intermediate/
total 858
-rw------- 1 cgregg operator 2279 May 29 09:29 00001.00000.mapped
-rw------- 1 cgregg operator 1448 May 29 09:29 00001.00001.mapped
-rw------- 1 cgregg operator 1927 May 29 09:29 00001.00002.mapped
-rw------- 1 cgregg operator 2776 May 29 09:29 00001.00003.mapped
-rw------- 1 cgregg operator 1071 May 29 09:29 00001.00004.mapped
...lots removed
-rw------- 1 cgregg operator  968 May 29 09:29 00012.00027.mapped
-rw------- 1 cgregg operator 1720 May 29 09:29 00012.00028.mapped
-rw------- 1 cgregg operator 1686 May 29 09:29 00012.00029.mapped
-rw------- 1 cgregg operator 2930 May 29 09:29 00012.00030.mapped
-rw------- 1 cgregg operator 2355 May 29 09:29 00012.00031.mapped
```

- If we look at **`00012.00028`**, we see:

```
myth57:$ head -10 files/intermediate/00012.00028.mapped
thee 1
rest 1
thee 1
woes 1
knows 1
grieve 1
sire 1
laertes 1
sire 1
power 1
```

- This file represents the words in **`00012.input`** that hashed to 28 modulo 32 (because we have 8 mappers * 4 reducers)
- Note that some words will appear multiple times (e.g., "thee")

MapReduce Assignment: Hashing

- If we look at `00005.00028`, we can also see "thee" again:

```
myth57:$ head -10 files/intermediate/00005.00028.mapped
vain 1
must 1
strand 1
cry 1
herself 1
she 1
along 1
head 1
dayreflection 1
thee 1
```

- This makes sense because "thee" also occurs in file `00005.input` (these files are not reduced yet!)
- "thee" hashes to 28 modulo 32, so it will end up in any of the `.00028` files if occurs in the input that produced that file.
- To test a word with a hash, you can run the hasher program, located here.

```
myth57:$ ./hasher thee 32
28
```

MapReduce Assignment: Starter Code

- Let's test the starter code (this only runs map):

```
myth57:~$ make directories filefree
// make command listings removed for brevity
myth57:~$ make
// make command listings removed for brevity
myth57:~$ ./mr --mapper ./mrm --reducer ./mrr --config odyssey-full.cfg --map-only --quiet
/afs/ir.stanford.edu/users/c/g/cgregg/assign8/files/intermediate/00001.mapped hashes to 2579744460591809953
/afs/ir.stanford.edu/users/c/g/cgregg/assign8/files/intermediate/00002.mapped hashes to 15803262022774104844
/afs/ir.stanford.edu/users/c/g/cgregg/assign8/files/intermediate/00003.mapped hashes to 15899354350090661280
/afs/ir.stanford.edu/users/c/g/cgregg/assign8/files/intermediate/00004.mapped hashes to 15307244185057831752
/afs/ir.stanford.edu/users/c/g/cgregg/assign8/files/intermediate/00005.mapped hashes to 13459647136135605867
/afs/ir.stanford.edu/users/c/g/cgregg/assign8/files/intermediate/00006.mapped hashes to 2960163283726752270
/afs/ir.stanford.edu/users/c/g/cgregg/assign8/files/intermediate/00007.mapped hashes to 3717115895887543972
/afs/ir.stanford.edu/users/c/g/cgregg/assign8/files/intermediate/00008.mapped hashes to 8824063684278310934
/afs/ir.stanford.edu/users/c/g/cgregg/assign8/files/intermediate/00009.mapped hashes to 673568360187010420
/afs/ir.stanford.edu/users/c/g/cgregg/assign8/files/intermediate/00010.mapped hashes to 9867662168026348720
/afs/ir.stanford.edu/users/c/g/cgregg/assign8/files/intermediate/00011.mapped hashes to 5390329291543335432
/afs/ir.stanford.edu/users/c/g/cgregg/assign8/files/intermediate/00012.mapped hashes to 13755032733372518054
myth57:~$
```

- If we look in **`files/intermediate`**, we see files without the reducer split:

```
myth57:~$ $ ls -l files/intermediate/
total 655
-rw------- 1 cgregg operator 76280 May 29 10:26 00001.mapped
-rw------- 1 cgregg operator 54704 May 29 10:26 00002.mapped
-rw------- 1 cgregg operator 53732 May 29 10:26 00003.mapped
-rw------- 1 cgregg operator 53246 May 29 10:26 00004.mapped
-rw------- 1 cgregg operator 53693 May 29 10:26 00005.mapped
-rw------- 1 cgregg operator 53182 May 29 10:26 00006.mapped
-rw------- 1 cgregg operator 54404 May 29 10:26 00007.mapped
-rw------- 1 cgregg operator 53464 May 29 10:26 00008.mapped
-rw------- 1 cgregg operator 53143 May 29 10:26 00009.mapped
-rw------- 1 cgregg operator 53325 May 29 10:26 00010.mapped
-rw------- 1 cgregg operator 53790 May 29 10:26 00011.mapped
-rw------- 1 cgregg operator 52207 May 29 10:26 00012.mapped
```

- It turns out that "thee" is only in 11 of the 12 files:

```
$ grep -l "^thee " files/intermediate/*.mapped \
         | wc -l
11
```

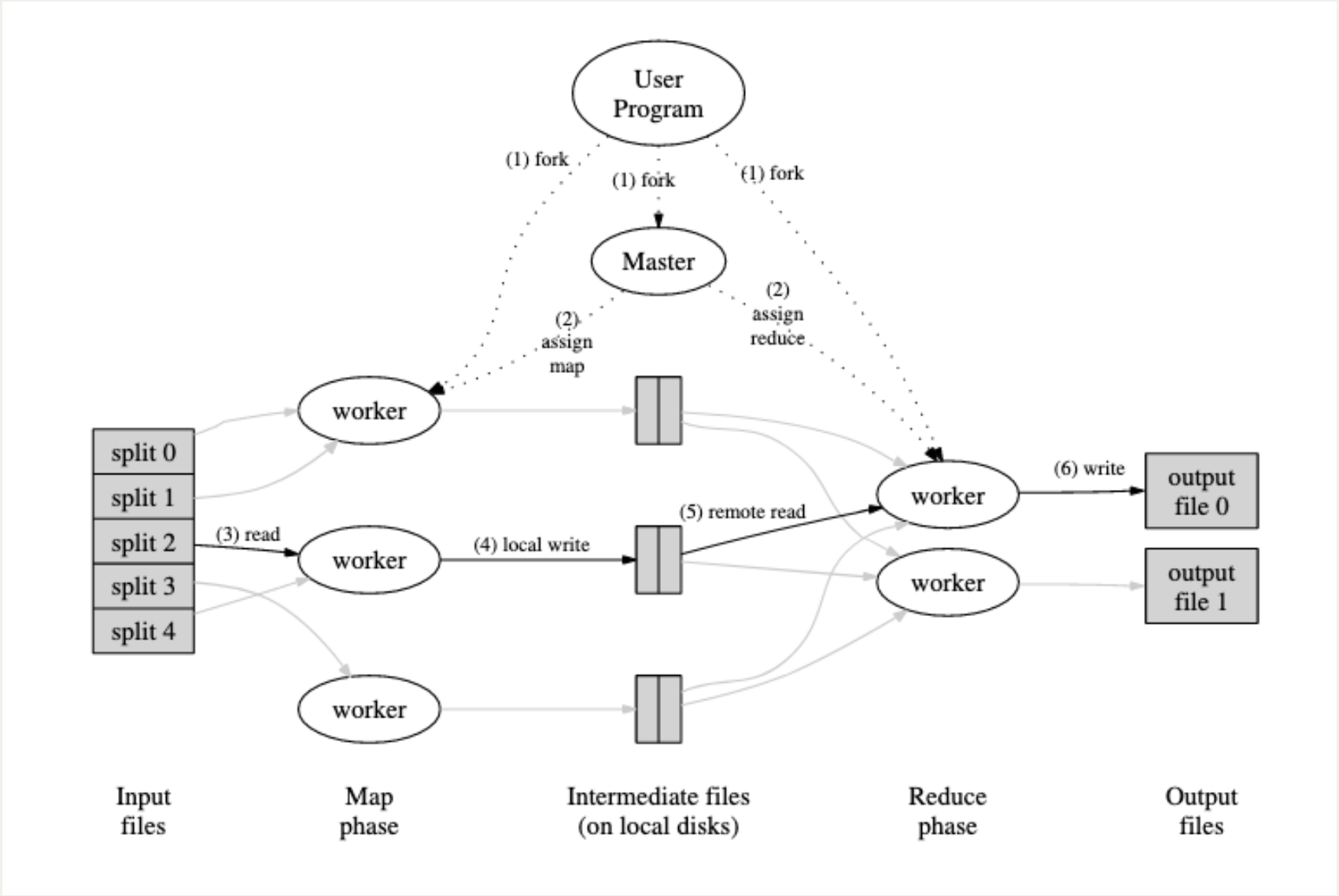
- also, **`files/output`** is empty:

```
myth57:$ ls -l files/output/
total 0
```

MapReduce questions

Centralized vs. Distributed

- Master sends map and reduce tasks to workers



Centralized vs. Distributed

- Master sends map and reduce tasks to workers
  - This gives master a global view of the system: progress of map and reduce tasks, etc.
  - Simplifies scheduling and debugging
- Master can become a bottleneck: what if it cannot issue tasks fast enough?
  - Centralized controllers can usually process 8,000-12,000 tasks/second
  - MapReduce generally does not hit this bottleneck
    - Both map and reduce tasks read from disk: take seconds-tens of seconds
    - MapReduce can scale to run on 80,000 - 120,000 cores
  - More modern frameworks, like Spark, can
    - Spark and other frameworks operate on in-memory data
    - Significant work to make them computationally fast: sometimes 10s of ms

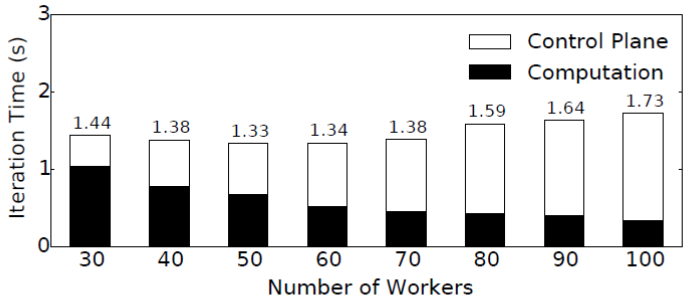


Figure 1: The control plane is a bottleneck in modern analytics workloads. Increasingly parallelizing logistic regression on 100GB of data with Spark 2.0's MLlib reduces computation time (black bars) but control overhead outstrip these gains, *increasing* completion time.

"Execution Templates: Caching Control Plane Decisions for Strong Scaling of Data Analytics"

Omid Mashayekhi, Hang Qu, Chinmayee Shah, Philip Levis

In Proceedings of 2017 USENIX Annual Technical Conference (USENIX ATC '17)

Centralized Bottleneck

- Graph shows what happens as you try to parallelize a Spark workload across more servers: it slows down
- More and more time is spent in the "control plane" -- sending messages to spawn tasks to compute
- Nodes fall idle, waiting for the master to send them work to do.
- Suppose tasks are 100ms long
  - 30 workers = 2400 cores
  - Can execute 24,000 tasks/second
  - Master can only send 8,000

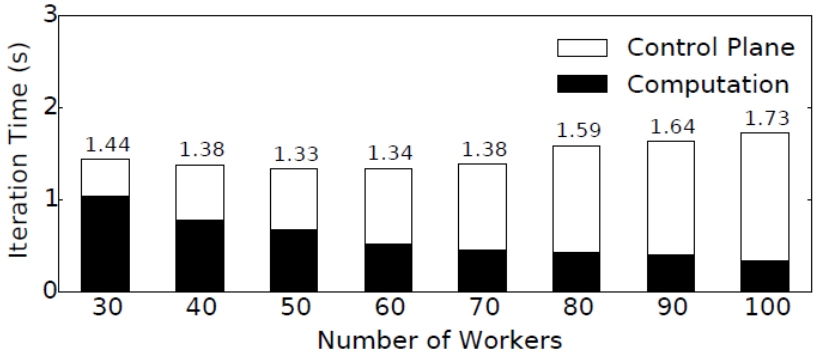


Figure 1: The control plane is a bottleneck in modern analytics workloads. Increasingly parallelizing logistic regression on 100GB of data with Spark 2.0's MLlib reduces computation time (black bars) but control overhead outstrip these gains, *increasing* completion time.

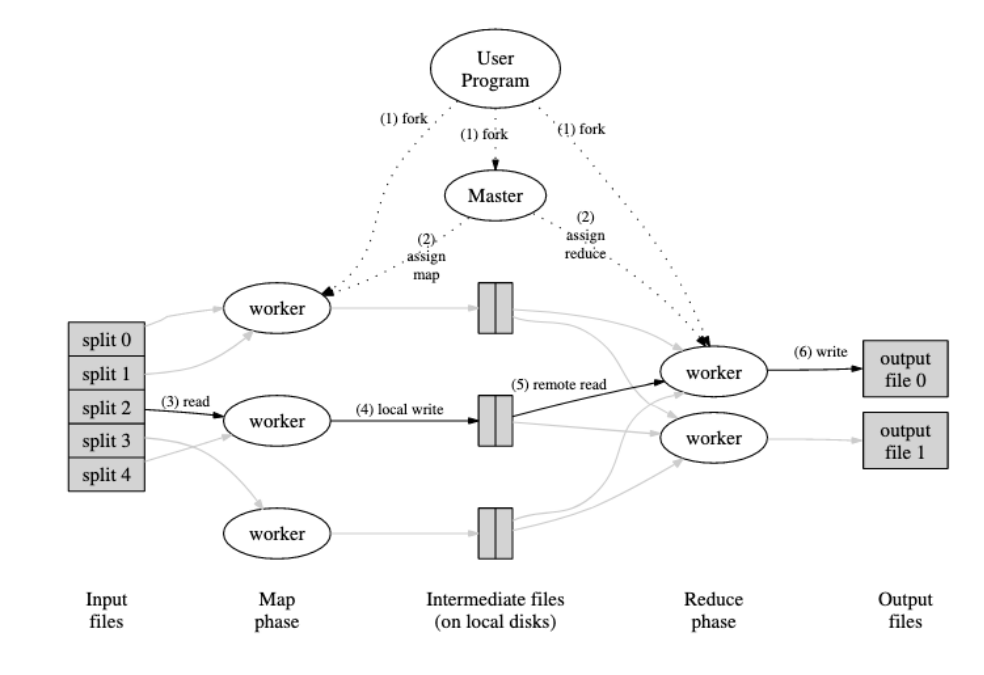
- This bottleneck will only worsen as we make tasks computationally faster: Spark used to be 50x slower than C++, now it is 6x slower (3x for JVM, 2x for data copies)
  - What happens when it involves C++ libraries and is 1.1x slower?
  - What happens when it invokes GPUs?

Decentralized Approach (e.g., Naiad, MPI)

- Some systems have no centralized master: there are only worker nodes
- Problem: what do you do when one of the workers fails? How do other workers discover this and respond appropriately?
  - In practice, these systems just fail and force you to restart the computation
  - Complexity of handling this is not work the benefit of automatic recovery
- Problem: how do you load balance?
  - You do so locally and suboptimally or not at all
  - Just try to keep workers busy
  - Work stealing: if a worker falls idle, tries to steal work from adjacent workers
  - Can require complex reconfiguration of program, as workers need to know new placements of computations/data (e.g., Naiad requires installing new dataflow graph)

Idea: Cache Control Plane Messages

- If a job uses fast tasks and runs for a long time, it must either be an enormously long program or have loops
- Loops are repeated patterns of control plane messages: they execute the same tasks again and again
  - Common in machine learning, simulations, data analytics, approximations, etc.
- Idea: rather than execute each loop iteration from scratch, *cache* a block of control plane operations and re-instantiate with a single message
- This occurs between the user program and the master as well as between the master and workers
- Call this cached structure a *template:* some values and structures are static (like which tasks to execute), others are bound dynamically (like which data objects to operate on, task IDs, etc.)

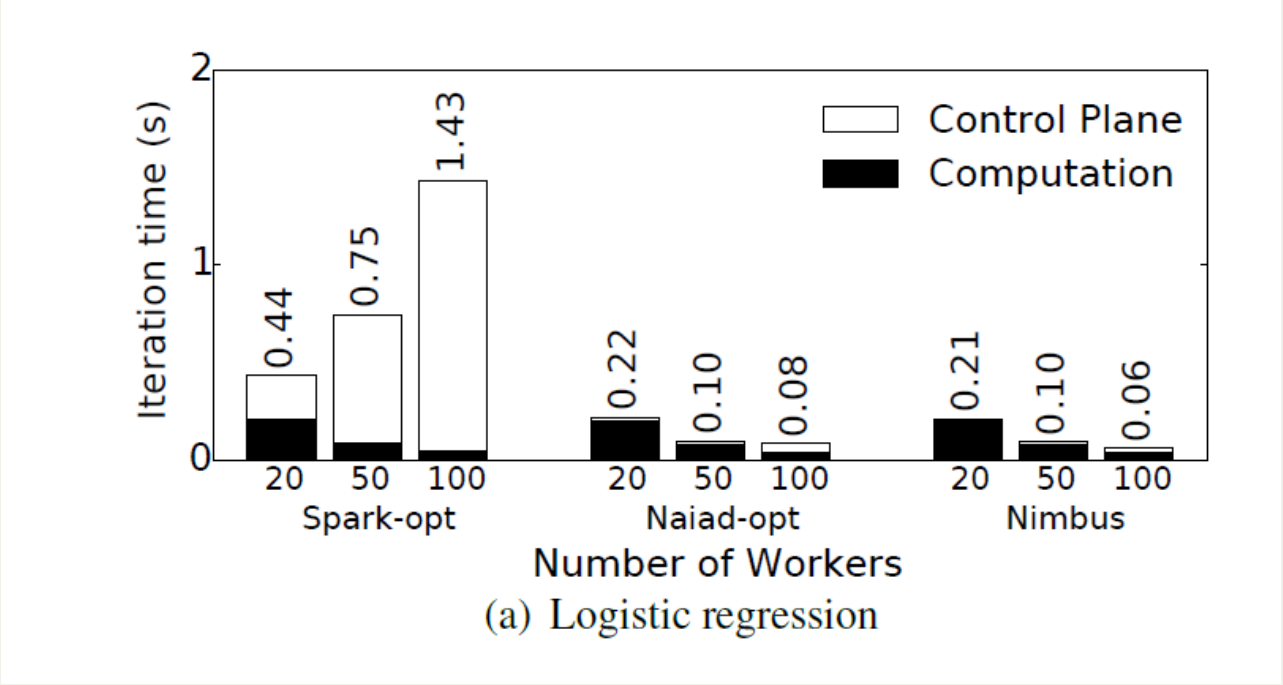


"Execution Templates: Caching Control Plane Decisions for Strong Scaling of Data Analytics"

Omid Mashayekhi, Hang Qu, Chinmayee Shah, Philip Levis

In Proceedings of 2017 USENIX Annual Technical Conference (USENIX ATC '17)

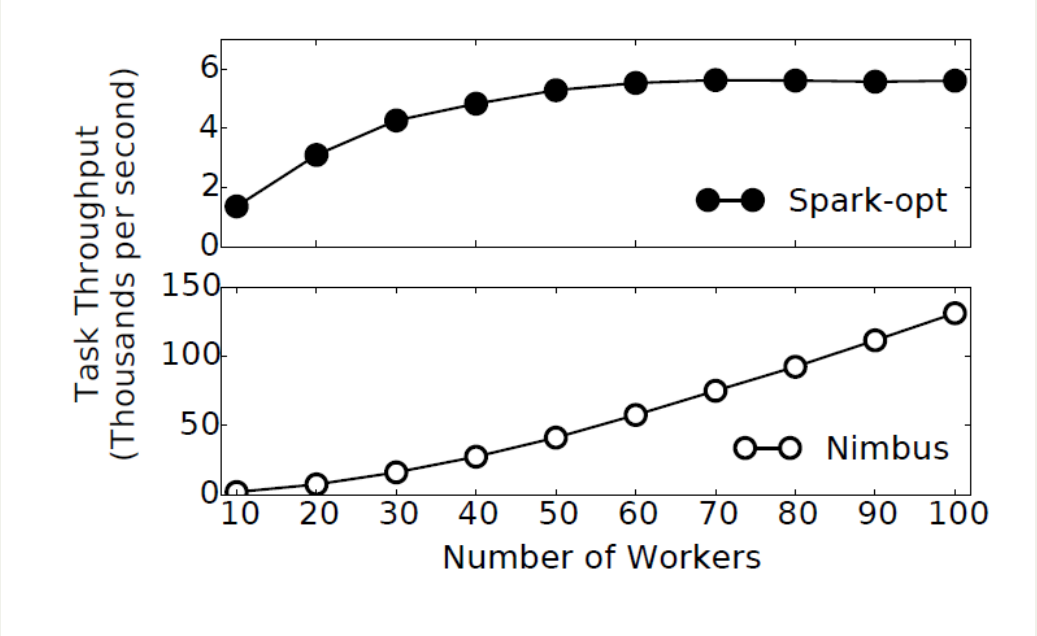
Results



- Caching control plane messages (Nimbus) is as fast as systems that do not have a centralized master (Naiad-opt), and much faster than systems which have a centralized master but do not cache messages (Spark-opt)

"Execution Templates: Caching Control Plane Decisions for Strong Scaling of Data Analytics"
Omid Mashayekhi, Hang Qu, Chinmayee Shah, Philip Levis
In Proceedings of 2017 USENIX Annual Technical Conference (USENIX ATC '17)

Results



- Caching control plane messages allows Nimbus to scale to support over 100,000 tasks/second, while Spark bottlenecks around 6,000

"Execution Templates: Caching Control Plane Decisions for Strong Scaling of Data Analytics"
Omid Mashayekhi, Hang Qu, Chinmayee Shah, Philip Levis
In Proceedings of 2017 USENIX Annual Technical Conference (USENIX ATC '17)

Results

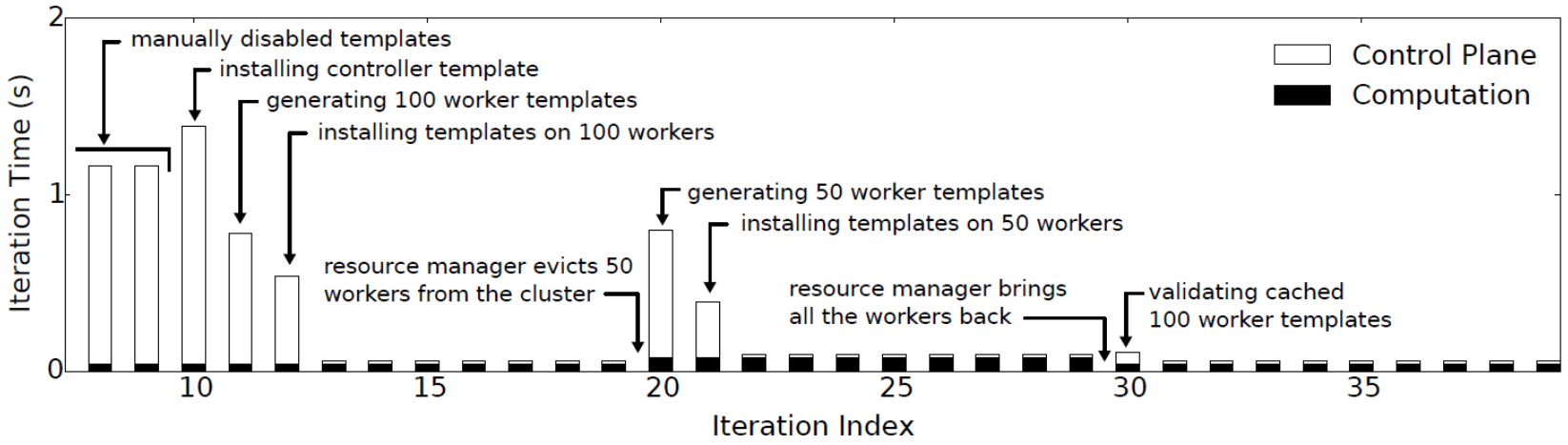


Figure 10: Execution templates can schedule jobs with high task throughputs while dynamically adapting as resources change. This experiment shows the control overheads as a cluster resource manager allocates 100 nodes to a job, revokes 50 of the nodes, then later returns them.

- Because systems caching control messages can use a centralized manager, they can recover from failures and load-balance easily, although doing so requires generating new templates and caching them.

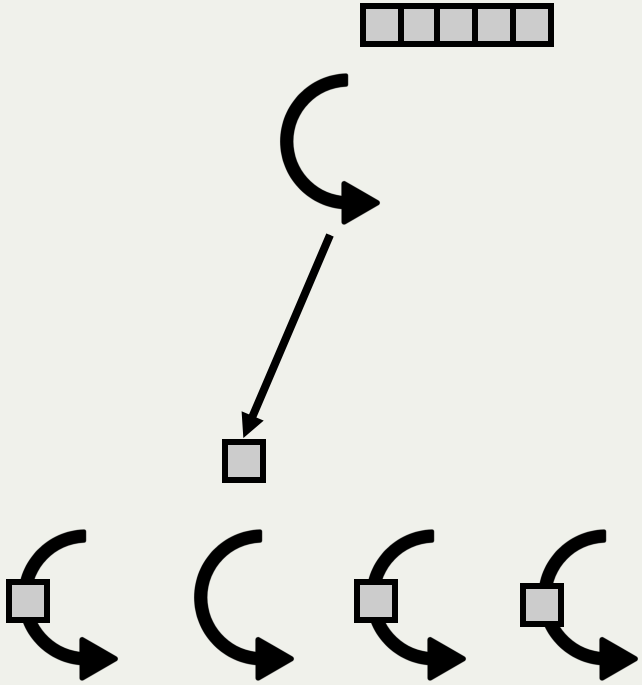
"Execution Templates: Caching Control Plane Decisions for Strong Scaling of Data Analytics"
Omid Mashayekhi, Hang Qu, Chinmayee Shah, Philip Levis
In Proceedings of 2017 USENIX Annual Technical Conference (USENIX ATC '17)

Question From Piazza

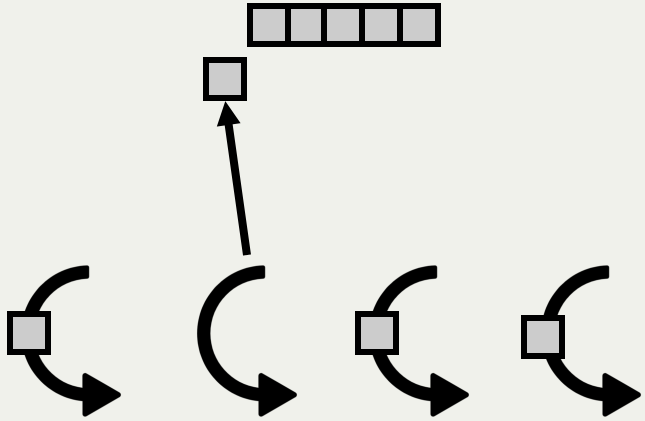
The assignment spec suggests that we should be able to select a thread to which we assign a thunk ("The dispatcher thread should loop almost interminably. In each iteration, it should sleep until schedule tells it that something has been added to the queue. It should then wait for a worker to become available, select the worker, mark it as unavailable, dequeue a function, put a copy of that function in a place where the worker can access it, and then signal the worker to execute it").

This requires bookkeeping at a fine granularity as we need to store context for each thread. An alternative implementation would be to keep a global task queue and a synchronization mechanism realized by, say, a single condition variable so that we can simply `nofity_one` just one thread from the pool of worker threads and let it execute the thunk. This should work just fine as long as we do not care which thread performs the task, and that seems to be the case in this assignment. This alternative implementation is likely more lightweight, since we will be freed from keeping track of each thread's context. Maybe I am missing something, but what are the reasons for having the control over exactly which thread executes a thunk?

Visually



Centralized with dispatcher thread



Distributed: workers pull thunks off queue