SE3331-1 (2023 Fall)

Distributed Computing frameworks

Batch processing systems

Xingda Wei, Yubin Xia

IPADS, Shanghai Jiao Tong University

https://www.sjtu.edu.cn

Credits: Rong Chen@IPADS, CMU15-418

Review: (Approximate) computation for NN like ChatGPT

Cost of training ~=

Known fact: GPT-4 has 1.76 trillion parameters [1]

- This is 1,760,000,000,000 parameters
- So, this is 10,560,000,000,000 calculations for a single input of **a single iteration**!!

What are the computation capabilities of nowadays devices (e.g., A100)?

- 19.5 TFLOPS = 19,500,000,000,000 (FP32) float point per second
- Basically, it needs 30 seconds for an A100 GPU to finish an iteration **in the optimal case**

We need a powerful computation device for the AI!

[1] https://the-decoder.com/gpt-4-has-a-trillion-parameters/

Review: parallelism on a single device

3 parallelism strategies on a single device

- Single core+: pipeline + super scalar with instruction level parallelism (ILP)
- Single core++: added SIMD support
- Multiple core: a single core (single core, single core+, single core++) can be glued together !

Question: what's next? Single core+++++?

- Solution (Out of the scope of this lecture): domain-specific accelerators !

Single core+++++: domain-specific accelereators

The era of domain-specific accelerators

Accelerators (even on general-purpose computing devices)

- Hardware designed to fulfill a single task
- Typically are not general-purpose, e.g., not programmable

CPU:

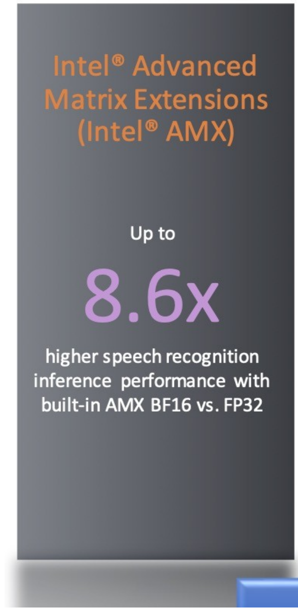
- SIMD + Matrix accelerators (e.g., Intel's AMX accelerators)

GPU:

- Tensorcore: accelerators for matrix operations

TPU (Tensor processing unit):

- tensor process (optimized for large matrix operations)



AI Chip Landscape

Tech Giants/Systems
- Google
- Microsoft
- aws
- IBM
- facebook
- HUAWEI
- Bai du 百度
- Alibaba Group 阿里巴巴集团
- FUJITSU
- NOKIA
- TOSHIBA
- Hewlett Packard Enterprise
- DELL

IC Vender/Fabless
- intel
- SAMSUNG
- NVIDIA
- QUALCOMM
- AMD
- NXP
- ST
- XILINX
- MEDIATEK
- BROADCOM
- MARVELL
- Rockchip 瑞芯微电子

IP/Design Service
- arm
- synopsys
- Imagination
- cadence
- CEVA
- VeriSilicon
- alchip
- GUC
- FARADAY
- eSilicon

Startup in China
- Cambricon 寒武纪科技
- 地平线 Horizon Robotics
- BITMAIN
- intellifusion 云天励飞
- ChipIntelli
- Think Force
- Canaan
- 云知声 Unisound
- AISPEECH 思必驰 专注人性化的智能语音
- Rokid
- 肇观电子 NextVPU
- Enflame

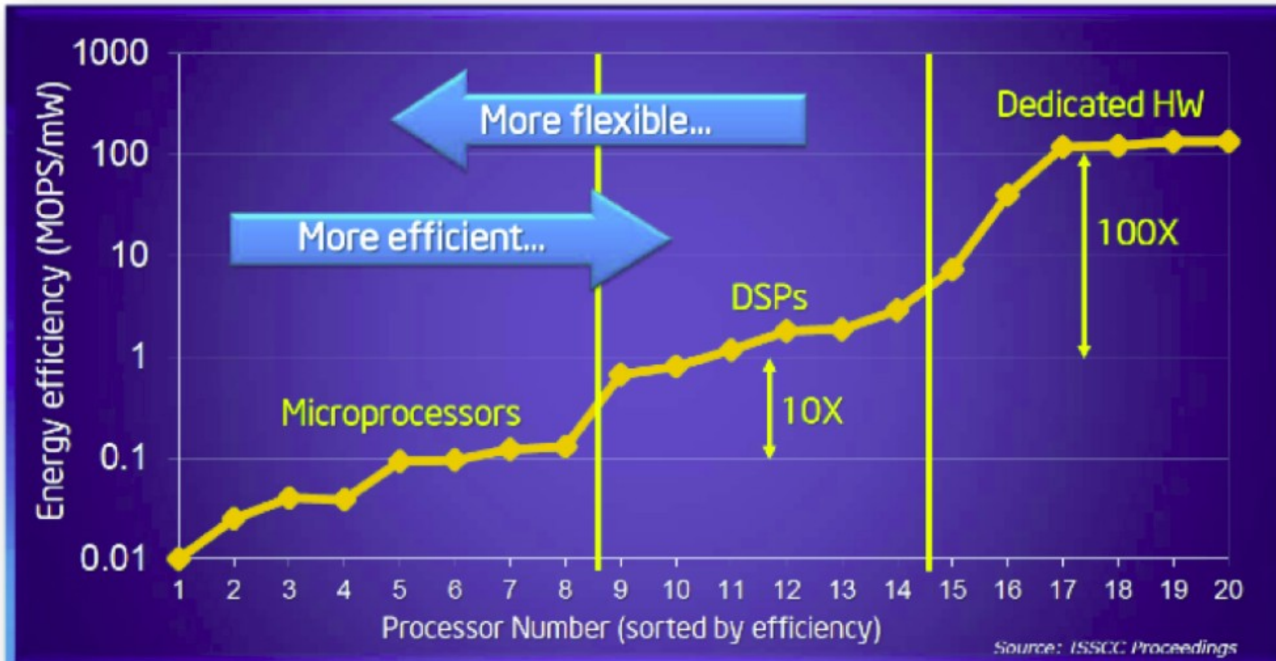
Startup Worldwide
- cerebras
- Wave Computing
- Graphcore
- habana
- SambaNova SYSTEMS
- thinci
- LIGHTELLIGENCE
- HAILO Empowering Intelligence
- KALRAY
- Tenstorrent
- MYTHIC
- Preferred Networks
- brainchip
- PEZY Computing
- GREENWAVES TECHNOLOGIES
- AIMOTIVE
- KONIKU
- Tachyum
- flexlogix
- SYNTIANT
- gyrfalcon technology
- NOVUMIND

more on https://basicmi.github.io/AI-Chip/

Compiler
- TensorFlow
- PyTorch
- NVIDIA TensorRT
- tvm
- nGraph Compiler stack (Beta)
- plaidML

Benchmarks
- MLPerf
- AI - Benchmark
- AI Matrix.
- AIIA

Accelerators: Flexibility vs. Efficiency Tradeoffs

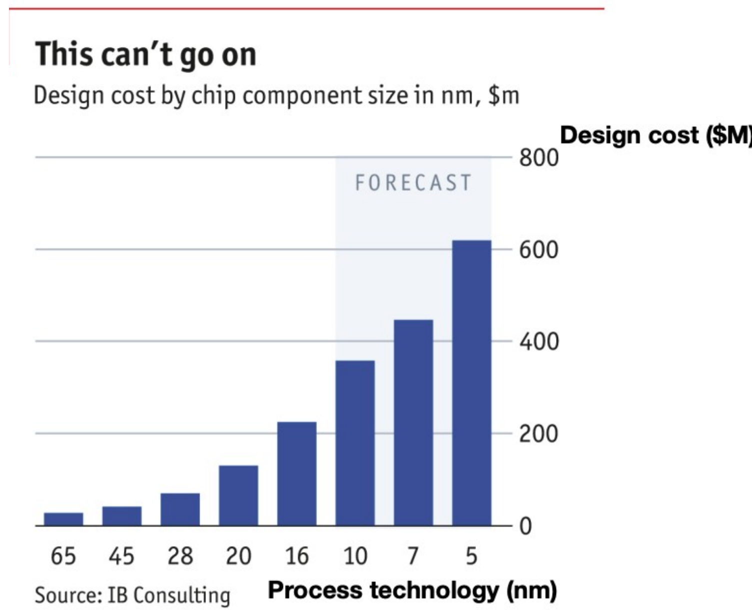


Source: Bob Broderson, Berkeley Wireless group

Specialization Challenge

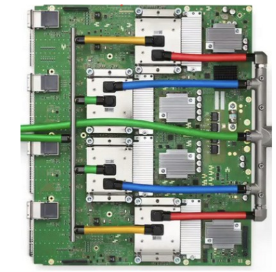
Tape-out costs for accelerators ASICs is exorbitant 10x cost gap between 16nm and 65nm

– Risky bet to design hardware accelerators for ever-changing applications



Review: Spectrum of computation device available

Programmability



Intel i5-9600K, single core: 6.3GFLOPS Multiple cores: 37.7GFLOPS

Mate60 GPU 2.3 TFLOPs

Apple M2 GPU 3.6 TFLOPs

NVIDIA A100 GPU 19.5 TFLOPs

Google TPUv4 275 TFLOPS

Performance

Remaining question: is a single device sufficient?

A device has limited physical capacity to store "cores" (chip size)

- Our cores are generalized, e.g., can either be CPU cores, GPU cores (cores w/o cache coherence + many SIMD ALUs, etc.), domain-specific cores

How to make a single device faster?

- Increasing clock rate (has a limit)
- Put more cores on a single chip, but has physical limits, e.g., chip size

Question: is a single device sufficient?

A device has limited physical capacity to store "cores" (chip size)

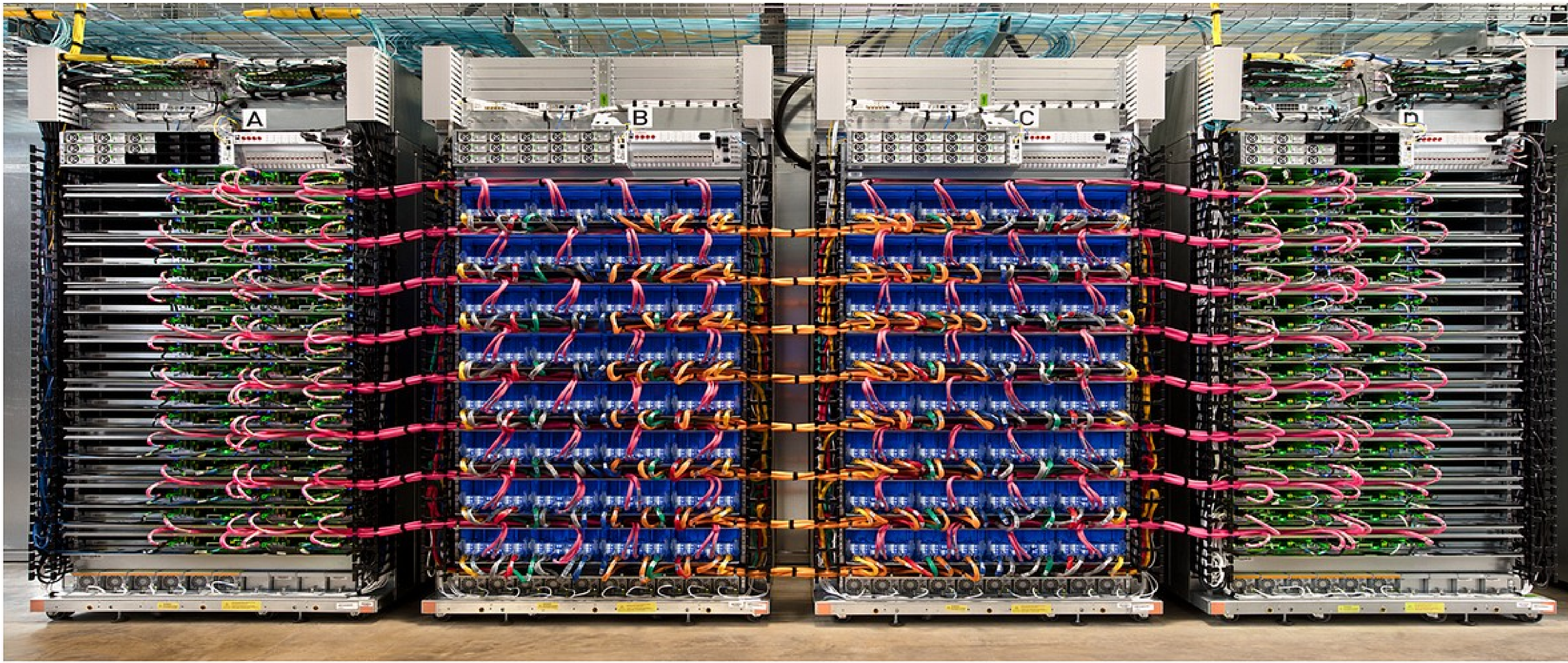
- Our cores are generalized, e.g., can either be CPU cores, GPU cores (cores w/o cache coherence + many SIMD ALUs, etc.), domain-specific cores

Why? Recall our previous calculation

- Basically, a A100 needs 30 seconds for an A100 GPU to finish an iteration on a single input (a.k.a, token) in the optimal case
- How many tokens are trained? 13T tokens! [1]
- To use one A100 to train GPT-4, we need about 412 years to finish the training

[1] https://the-decoder.com/gpt-4-has-a-trillion-parameters/

The case for distributed computing



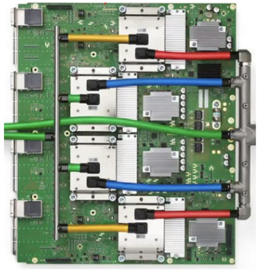
Example: Google's TPU v4 cluster

Spectrum of computation device available

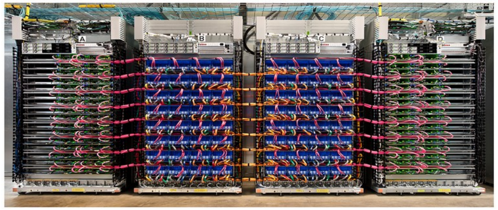
GPU
OPs

Apple M2
GPU 3.6 TFLOPs

NVIDIA A100
GPU 19.5 TFLOPs



Google TPUv4
275 TFLOPS



A cluster
? TFLOPS

Performance

Not so easy: many tedious things to cope with

Use distributed computing frameworks for complex queries

Each fits a common application scenario

- One sit (typically) does not fit all!

Batch processing system

- Spark, Hadoop, etc.

Today’s focus

Graph processing system

- GraphLab, etc.

Machine learning system

- TensorFlow, Pytorch, etc.

Note that we can also use one for another

- At the cost of programming effort and performance

Why traditional distributed computing frameworks first?

We need weapons to cope with the AI workload!

– The fundamental system requirements & techniques are similar

Boss(Training LLM)

You need weapons (from existing frameworks)

You

Example: batching processing

Suppose our web server records every requests in a log file

– E.g., using nginx log format, a log record looks like this:

```
216.58.210.78 - - [27/Feb/2015:17:55:11 +0000]
"GET /css/typography.css HTTP/1.1" 200 3377
"http://martin.kleppmann.com/"
"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_5)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/40.0.2214.115
Safari/537.36"
```

Abstract format: `$remote_addr - $remote_user [$time_local] "$request" $status $body_bytes_sent "$http_referer" "$http_user_agent"`

– **Task (simple log analysis)**: how to find top five popular pages?

Old days: using command line to process the log

```
216.58.210.78 - - [27/Feb/2015:17:55:11 +0000]
"GET /css/typography.css HTTP/1.1" 200 3377
"http://martin.kleppmann.com/"
"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_5)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/40.0.2214.115 Safari/537.36"
```

Find top five most popular pages w/ command line tools

```
cat /var/log/nginx/access.log | ❶
  awk '{print $7}' | ❷
  sort             | ❸
  uniq -c          | ❹
  sort -r -n       | ❺
  head -n 5          ❻
```

Use command line tools for batch processing

Find top five most popular pages

```
cat /var/log/nginx/access.log | ❶
  awk '{print $7}' | ❷
  sort             | ❸
  uniq -c          | ❹
  sort -r -n       | ❺
  head -n 5          ❻
```

1. Read log file
2. Filter the line, pick the 7th token, i.e., `/css/typography.css`
3. Sort so that the same requests come together
4. Aggregate, with a counter for each line
5. Sort again (using the counter)
6. Print 1-5 items

Does batch processing with command line tools scale?

```
cat /var/log/nginx/access.log | ❶
  awk '{print $7}'          | ❷
  sort                      | ❸
  uniq -c                   | ❹
  sort -r -n                | ❺
  head -n 5                   ❻
```

Command line tools are mostly **single-threaded**, and **single-machine**

- ❶: scalability is restricted by the disk capacity
- ❷～❻: restricted by single-thread computing power
- ❷～❻: also restricted by the machine's DRAM capacity

Scale using GFS

Process to scale a computation to distributed computing

A large problem to solve, e.g., log processing, AI training

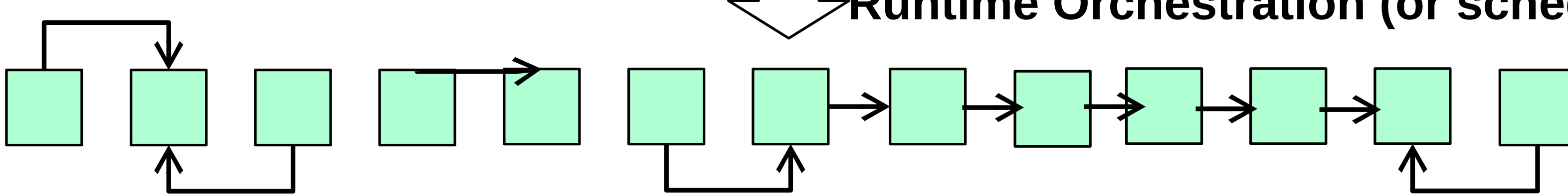
Decompose

Sub-problems. Many alias, e.g., sub-tasks, sub-jobs (or simply jobs)

Assignment

A set of physical programs that can run on parallel units, e.g., a thread or a GPU kernel

Runtime Orchestration (or scheduling)



Amdahl's Law revisited

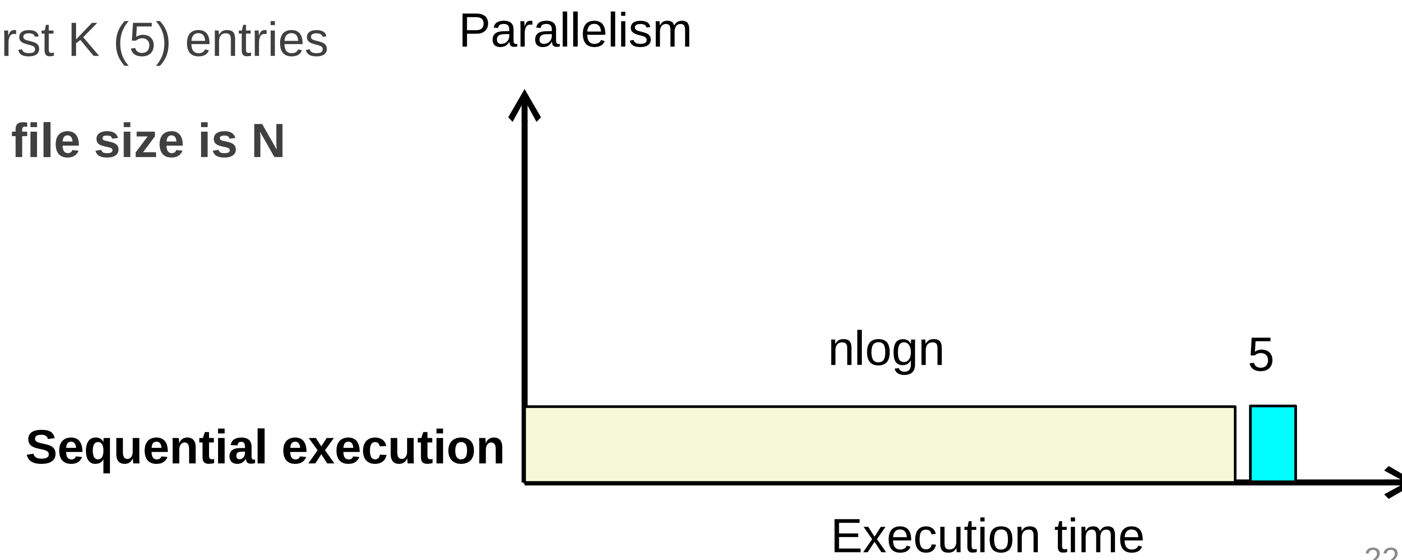
Let S = the fraction of sequential execution that is inherently sequential

- Then **maximum speedup** due to parallel execution **≤ 1/S**

Consider our previous log process example (simplified)

- Sort the input file
- Print the first K (5) entries

Suppose the file size is N

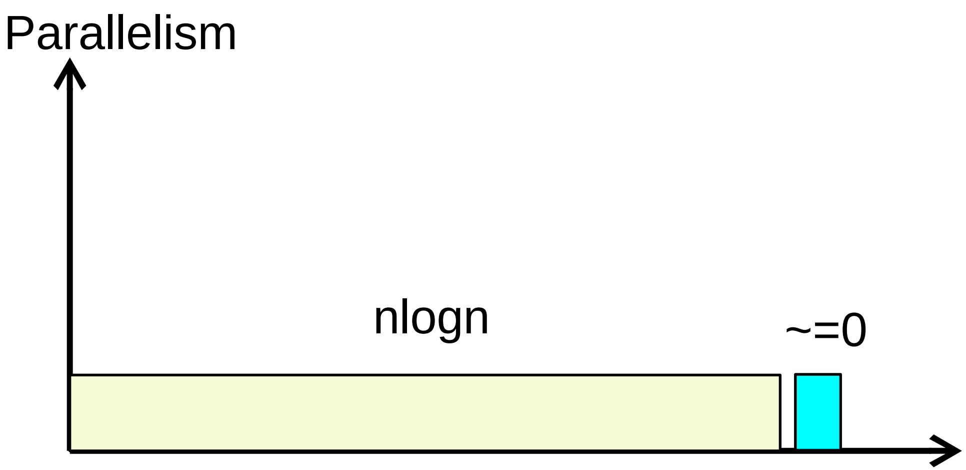


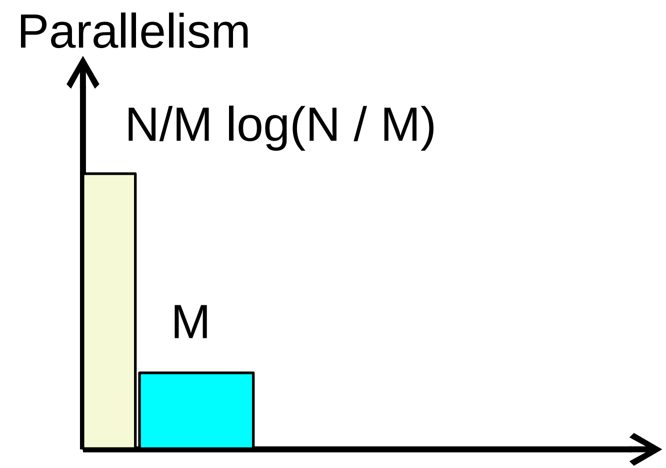
Parallelize with M machines (single-threaded)

Strategy (in theory)

– Step 1: decompose the file into M pieces, count top 5 concurrently

– Step 2: merge the top 5 from all the others & calculate the final top 5





Parallelize with M machines (single-threaded)

Strategy (in theory)

- Step 1: decompose the file into M pieces, count top 5 concurrently
- Step 2: merge the top 5 from all the others & calculate the final top 5

Speedup ~= M if N >> M

Speedup

But what if we want to calculate top-K?

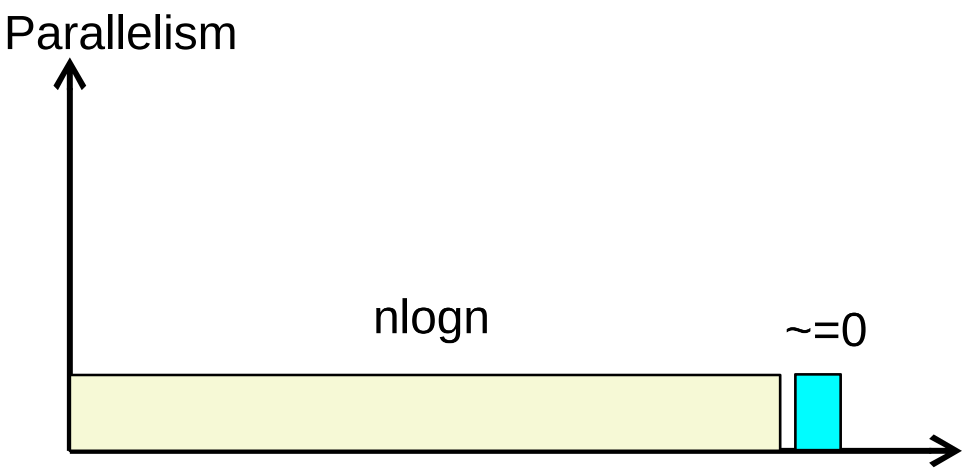
K is a relatively large number

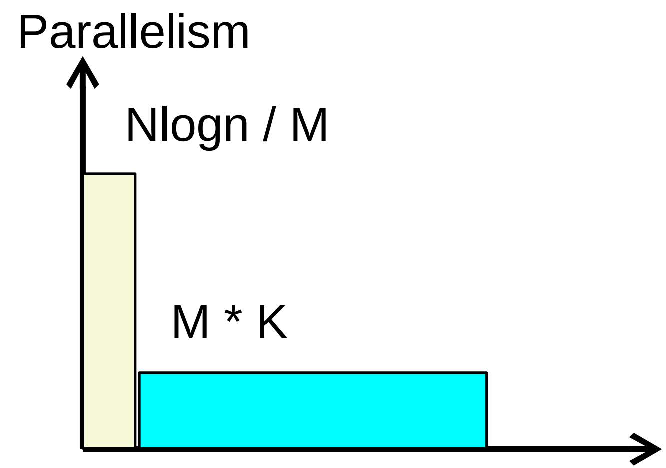
- E.g., very close to N
- If K grows, the speedup decreases

$$\frac{M}{1 + K \times \frac{M^2}{NlogN}}$$

Sequential part

There are optimization, but is more complex



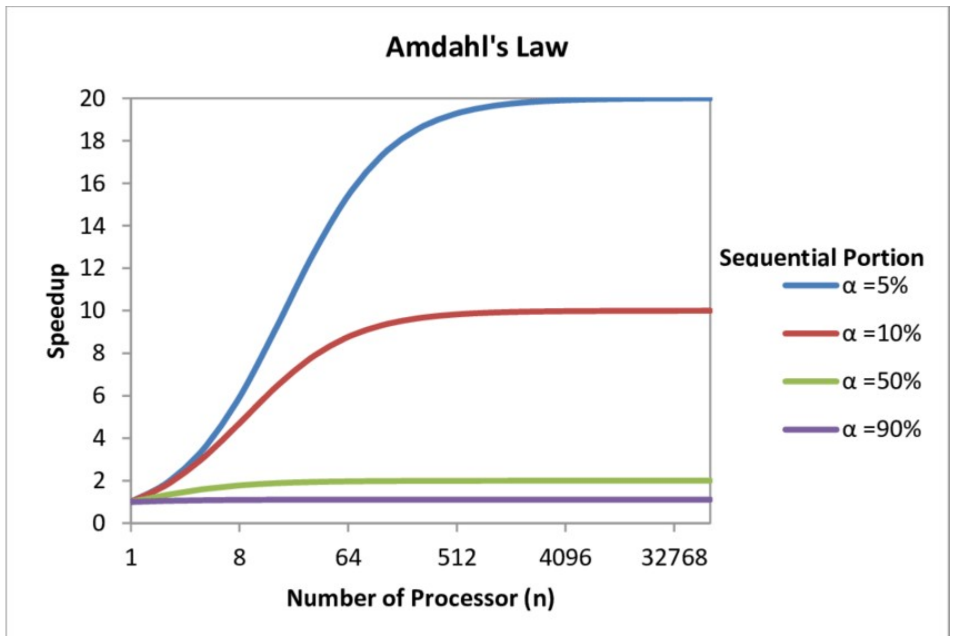


Amdahl's Law revisited

Let **= the fraction** **of total work that is** **inherently sequential**

Max speedup on M machines given by:

Speedup



Source: https://www.researchgate.net/figure/Graph-demonstrating-Amdahls-Law_fig2_315696585

From theory to reality

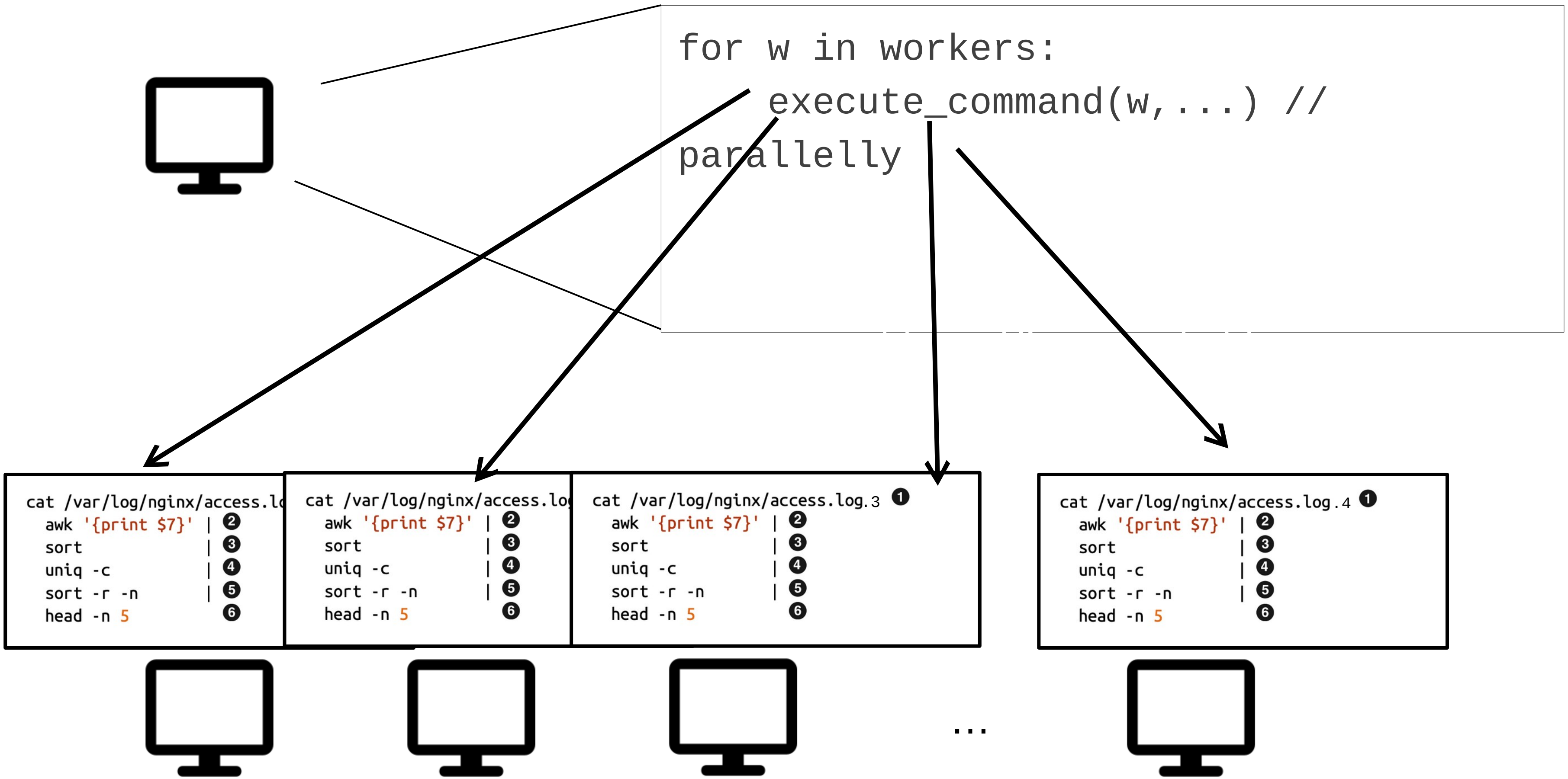
Strategy (in theory)

- Step 1: decompose the file into M pieces, count top 5 concurrently
- Step 2: merge the top 5 from all the others & calculate the final top 5

Step #1

- How to decompose? How to assign threads to do chunks?
- **Try #1:** ssh to M machines, run the program, each given a different file

Scaling log processing using ssh: step #1



From theory to reality

Strategy (in theory)

- Step 1: decompose the file into M pieces, count top 5 concurrently
- Step 2: merge the top 5 from all the others & calculate the final top 5

Step #1

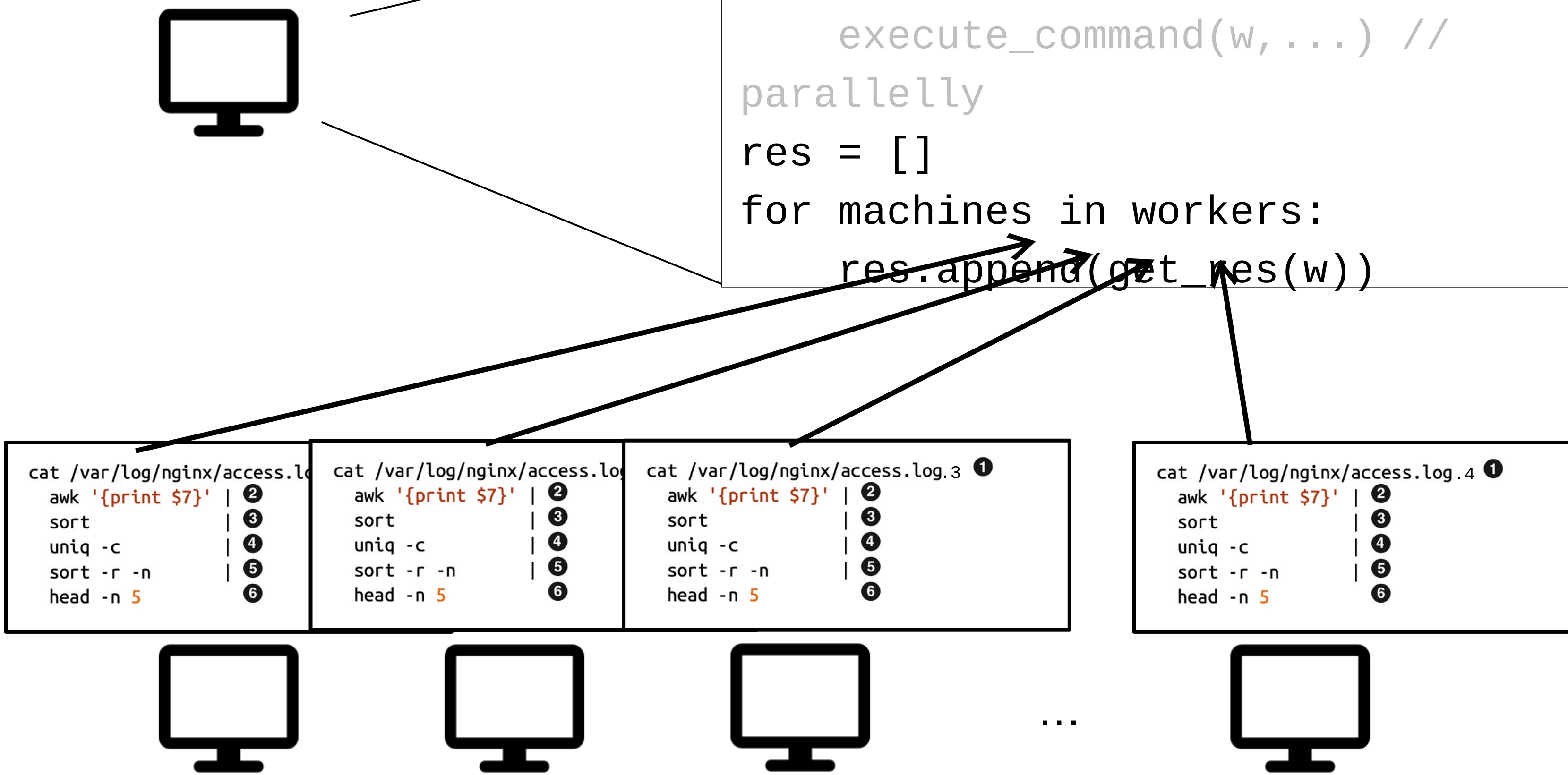
- How to decompose? How to assign threads to do chunks?
- **Try #1:** ssh to M machines, run the program, each given a different file

Step #2

- How can we collect the results from all the others?

Scaling log processing using RPC: step #2

```
for w in workers:
    execute_command(w,...) //
parallelly
res = []
for machines in workers:
    res.append(get_res(w))
```



Building application with distributed computing

Graduate Student

~~**ML experts**~~ repeatedly solve **the same** parallel design challenges

- Implement and debug complex parallel & distributed systems
- Tune for a specific parallel platform

6 months later, a conference paper contains:

"We implemented ____ in parallel."

However, the resulting **code** will be difficult to **maintain** and **extend**

couples computation task with distributed implementation

Challenges programmers facing

1. Sending **data** to/from nodes
2. **Coordinating** among nodes
3. Recovering from node **failure**
4. Optimizing for **locality**
5. Partition data to to enable more **parallelism**

Communicating between two machines can be simple, e.g., with RPC

*But, what about **10,000** machines?*

Challenges programmers facing

1. Sending **data** to/from nodes
2. **Coordinating** among nodes
3. Recovering from node **failure**
4. Optimizing for **locality**
5. Partition data to to enable more **parallelism**

When can we collect computing results from all the computing nodes?

Challenges programmers facing

1. Sending **data** to/from nodes
2. **Coordinating** among nodes
3. Recovering from node **failure**
4. Optimizing for **locality**
5. Partition data to to enable more **parallelism**

Failure is common, especially at large-scale

Challenges programmers facing

1. Sending **data** to/from nodes
2. **Coordinating** among nodes
3. Recovering from node **failure**
4. Optimizing for **locality**
5. Partition data to to enable m

Transferring data over network is usually much slower than local data access

Challenges programmers facing

1. Sending **data** to/from nodes
2. **Coordinating** among nodes
3. Recovering from node **failure**
4. Optimizing for **locality**
5. Partition data to to enable more **parallelism**

Reduce the bottleneck of sequential part

Idea: we build a framework to hide the above tasks

Goal

- Reduce programmer's effort in solving the above challenges

Challenges

- What are the abstraction? Thread & RPC is insufficient
- More general, harder to provide the above property
- E.g., if a thread fails, how can we know its progress?
    - It's a very hard problem, we will come back to it later

MapReduce: Simplified Data Processing on Large Clusters

MapReduce: a distributed batch processing system

Created by **Google** (OSDI'04)

– Jeffrey Dean and Sanjay Ghemawat

Inspired by LISP (function language)

– **Map** (function, set of values)
  - Applies function to each value in the set

```
(map #'length' (() (a) (a b) (a b c))) ✉ (0 1 2 3)
```

– **Reduce** (function, set of values)
  - Combines all the values using a function (e.g., +)

```
(reduce #'+' (1 2 3 4 5)) ✉ 15
```

MapReduce abstraction matches our requirements well

1. Sending data to/from nodes

   Data is sent between **Map** & **Reduce**

2. Coordinating among nodes

   Use the MapReduce semantic to coordinate

   - i.e.， only need to schedule the reducer after the mapper

3. Recovering from node failure

   Will be talked about later

4. Optimizing for locality

   Will be talked about later

5. Partition data to to enable more **parallelism**

   Map reduce enable arbitrary partition of the data

MapReduce Programming Model

Allows user to process *huge* amounts of data on *thousands* of nodes

Framework for **data-parallel** computing

- Programmers get **simple yet restricted** API

Users **don't** have to worry about handling:

- ~~parallelization~~
- ~~data distribution~~
- ~~load balancing~~
- ~~fault tolerance~~

MapReduce Programming Model

user

Functionality

Parallelization

Data Distribution

Fault Tolerance

Load Balance

MapReduce Programming Model

user

Functionality

Parallelization

MapReduce

Runtime

Load Balance

Two Primitives:

Map (***input***)

Reduce (***key***, ***values***)

MapReduce example： Word count

Word Count: Count # occurrences of each word in a collection of documents

A simplified process of our previous log analysis example

```
cat /var/log/nginx/access.log |  1
  awk '{print $7}' |             2
  uniq -c                        3
```

MapReduce example： Word count

Word Count: Count the occurrences number of each word in a collection of documents

Map:

– Parse data and emit each word with a count (1)

Reduce:

– Reduce: sum together counts each key (words)

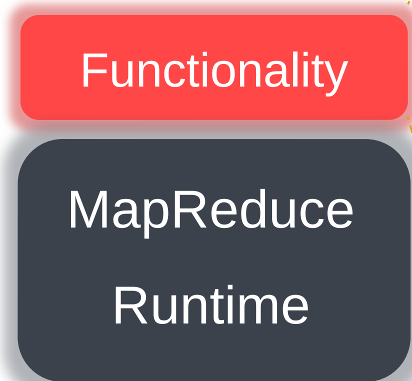
```
map (input) # a shard of file
        for each word w in input
                        EmitIntermediate (w,
        "1");
```

```
Reduce (String key, Iterator values)
        int result = 0;
        for each v in values
                result += ParseInt (v);
        emit (AsString (result));
```

MapReduce Programming Model

E.g., to find the most popular keywords (热词)

user



Two Primitive:

Map (***input***)
for each ***word*** in ***input***
emit (***word***, ***1***)

Reduce (***key***, ***values***)
int ***sum*** = 0;
for each ***value*** in ***values***
sum += ***value***;
emit (***word***, ***sum***)

MapReduce Programming Interface

Map: (input shard) ✉ intermediate (k/v pairs)

- Partition the input data into **M** “shards”
- Group all intermediate values associated with the same key
- Pass them to the **Reduce** function

Reduce: intermediate (k/v pairs) ✉ (results)

- Partition the key space into **R** “pieces” using a `Partition function`
    - E.g., hash(key) mod R (unsorted)
- Accept a key & a set of values
- Merge these values to form result of the key

MapReduce Execution Flow

Map

Grab the relevant data from the source
User function gets called for each chunk of input

Map Worker

Reduce Worker

Reduce

Aggregate the results
User function gets called for each unique key

MapReduce Execution Flow

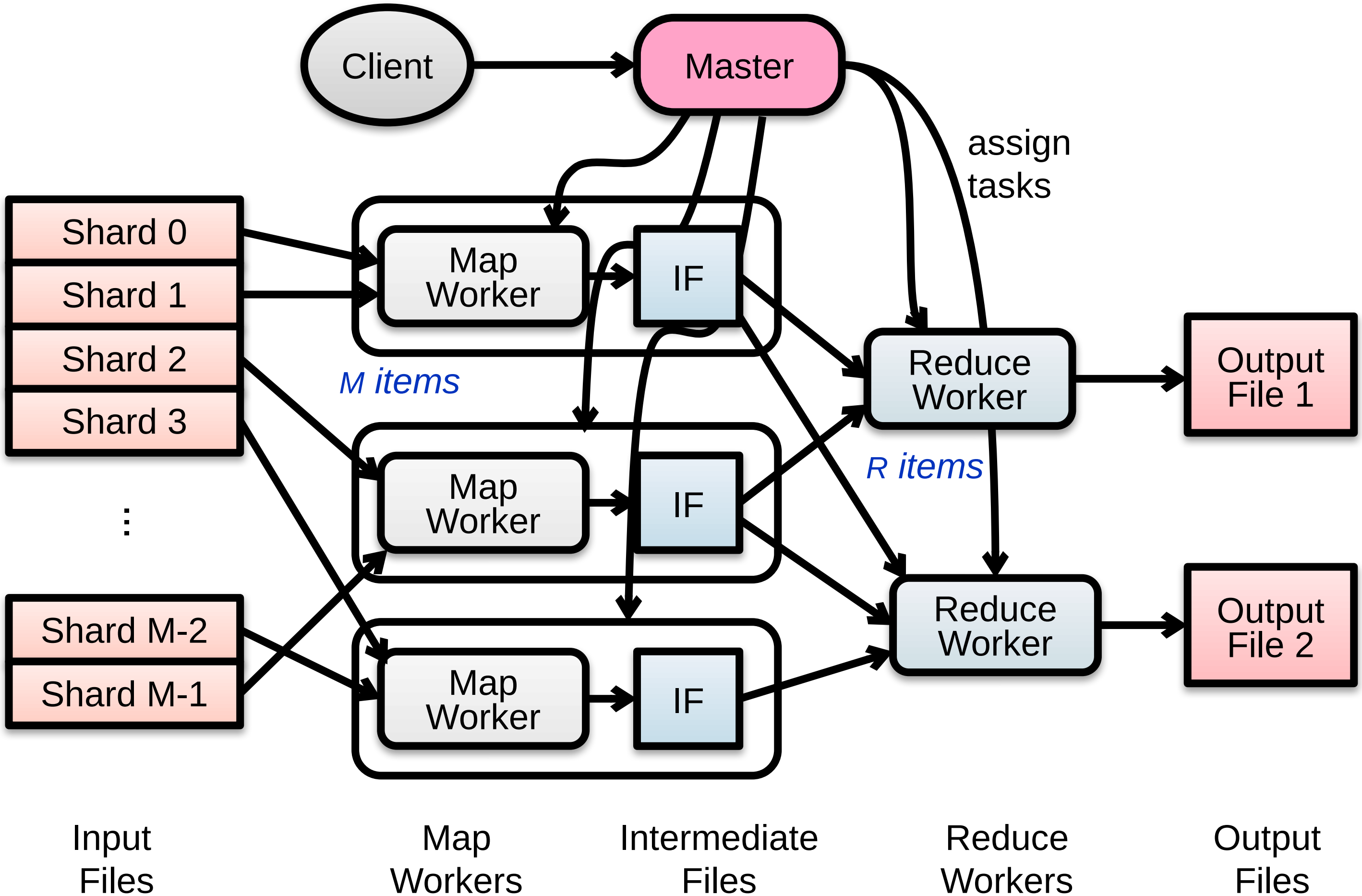
Map Worker

Map	Grab the relevant data from the source User function gets called for each chunk of input
Partition	Identify which Reducers will handle which keys Map partitions data to target Reducers

Reduce Worker

Sort	(Optional) Fetch the relevant partition data from all mappers Sort by keys
Reduce	Aggregate the results User function gets called for each unique key

MapReduce: The Complete Picture

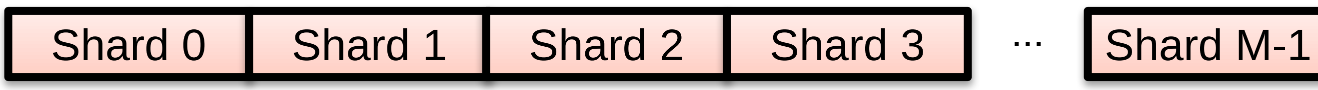


MapReduce: The Complete Picture

Step1: (Client/Master) split input files into chunks (shards)

– Typically, 64MB

– Why? Fit the GFS chunk size, so one RPC is sufficient to read the chunk



Input Files

Divided into M shards

MapReduce: The Complete Picture

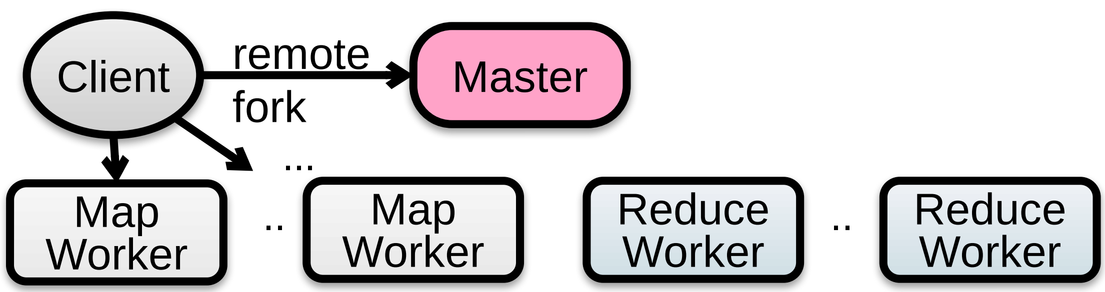
Step2: (remote) fork processes

Start up many copies of the program on cluster

- **1 master**: scheduler & coordinator
- Lots of **workers**

Idle workers are assigned either

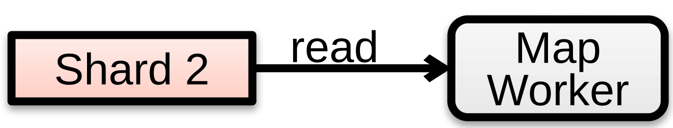
- Map tasks (each works on a shard)
- Reduce tasks (each works on intermediate files)



MapReduce: The Complete Picture

Step3: map task

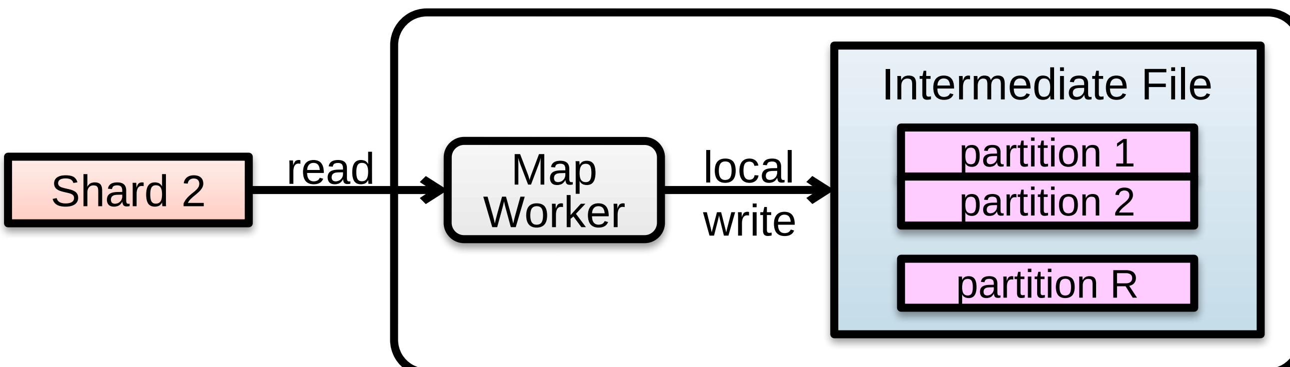
- Reads contents of the input **shard** assigned to it
- Parses **key/value** pairs out of the input data
- Passes each pair to a user-defined **`Map`** function
    - Produces **intermediate** key/value pairs
    - These are buffered in **memory**



MapReduce: The Complete Picture

Step4: create intermediate files

- Intermediate k/v pairs buffered in **memory** and periodically written to the **local disk**
  - Partitioned into **R** regions by a `Partition` function
- Notifies master when complete
  - Passes location of intermediate data to the **master**
  - **Master** forwards locations to the Reduce worker



MapReduce: The Complete Picture

Step4a: partitioning

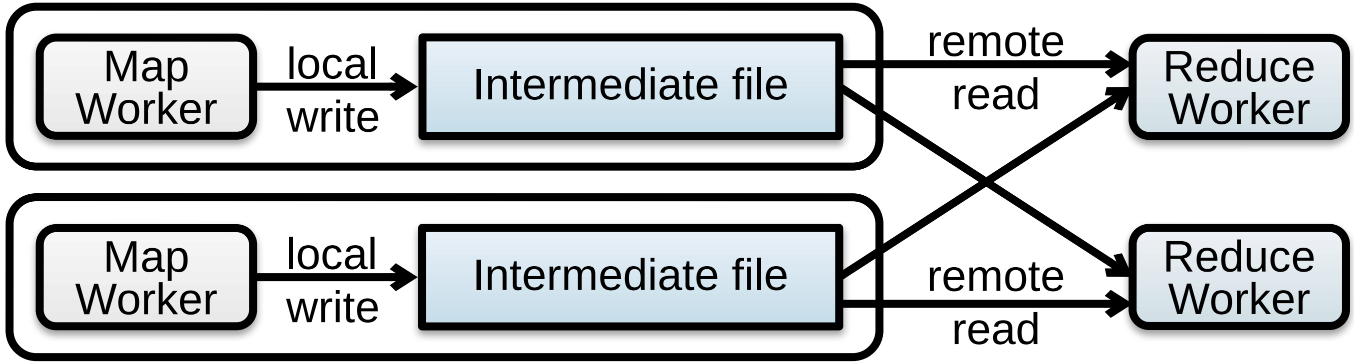
- **Map** data will be processed by reduce workers
    - The user's `Reduce` function will be called once per **unique key**
- Sort all the (key, value) data by keys
- `Partition` function: decides which of R reduce workers will work on which key
    - Default function: hash(key) mod R
    - Map worker partitions the data by keys
- Each reduce worker will read their partition from **every** map worker

MapReduce: The Complete Picture

Step5: sorting intermediate data

- Notified by master about the location of intermediate files for its partition
- Uses **RPCs** to read the data from the local disks of map workers
- **Sorts** the data by the intermediate keys

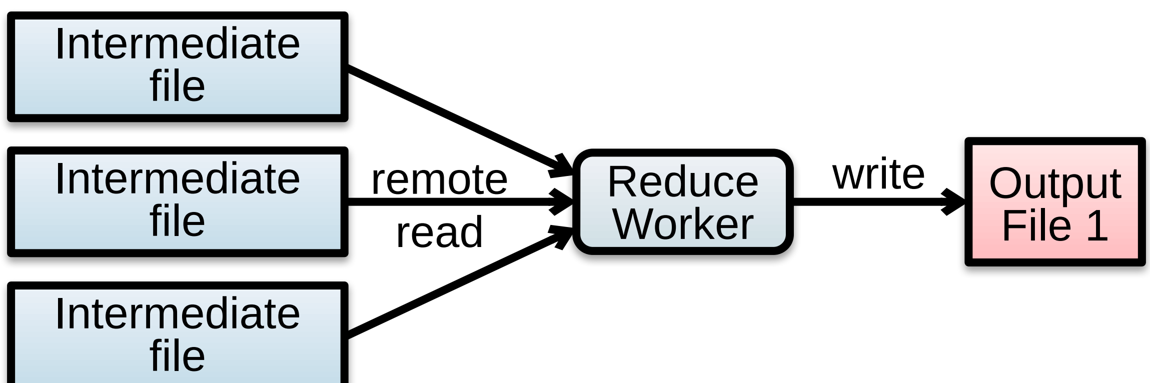
=> All occurrences of the same key are **grouped**



MapReduce: The Complete Picture

Step6: reduce task

- Groups data with a unique intermediate key
- User's `Reduce` function is given the key and the set of intermediate values for that key
    - <key, (value1, value2, ...)>
- The output is appended to an output file



MapReduce: The Complete Picture

Step7: return to user

- When all map and reduce tasks have completed
- Master wakes up the user program
- The **MapReduce** call in the user program returns, and the program can resume execution
    - Output of MapReduce is available in R output files

Recall MapReduce example： Word count

Word Count: Count # occurrences of each word in a collection of documents

Map:

– Parse data and emit each word with a count (1)

Reduce:

– Reduce: sum together counts each key (words)

```
map (input)
    for each word w in input
        EmitIntermediate (w, "1");
```

```
Reduce (String key, Iterator values)
    int result = 0;
    for each v in values
        result += ParseInt (v);
    emit (AsString (result));
```

MapReduce example re-visited： Word count

Input file

It will be seen that this mere painstaking burrower and grub-worm of a poor devil of a Sub-Sub appears to have gone through the long Vaticans and streetstalls of the earth, picking up whatever random allusions to whales he could anyways find in any book whatsoever, sacred or profane. Therefore you must not, in every case at least, take the higgledy-piggledy whale statements, however authentic, in these extracts, for veritable gospel cetology. Far from it. As touching the ancient authors generally, as well as the poets here appearing, these extracts are solely valuable or entertaining, as affording a glancing bird's eye view of what has been promiscuously …

Map

After Map
[Intermediate]

...
it 1
will 1
be 1
seen 1
that 1
this 1
mere 1
painstaking 1
burrower 1
and 1
grub-worm 1
of 1
a 1
poor 1
devil 1
of 1
a 1
sub-sub 1
…

After Sort

...
a 1
a 1
aback 1
aback 1
abaft 1
abaft 1
abandon 1
abandon 1
abandon 1
abandoned 1
abandoned 1
abandoned 1
abandoned 1
abandoned 1
abandoned 1
abandoned 1
abased 1
abased 1
…

Reduce

After Reduce

a 4736
aback 2
abaft 2
abandon 3
abandoned 7
abased 2
abasement 1
abashed 2
abate 1
abated 3
abatement 1
abating 2
abbreviate 1
abbreviation 1
abeam 1
abed 2
abednego 1
abel 1
abhorred 3
...

Other examples of MapReduce

Distributed grep

- *Search for words in lots of documents*
- **Map**: emit a line if it matches a given pattern
- **Reduce**: just output the intermediate data

Count URL access frequency

- *Find the frequency of each URL in web logs*
- **Map**: process logs of web page access;
  output <URL, 1>
- **Reduce**: add all values for the same URL

Other examples of MapReduce

Inverted index

- *Find what documents contain a specific word*
- **Map**: parse document, emit <word, doc-ID>
- **Reduce**: sort the doc-ID for each word, and output <word, list(doc-ID)>

Reverse web-link graph

- *Find where page links come from*
- **Map**: output <target, source> for each link to target in a page source
- **Reduce**: concatenate all source for each target, output <target, list(source)>

Other examples of MapReduce

Google **used** MapReduce to support distributed computing

- Large-scale machine learning jobs
- Large-scale graph analytic jobs
- Etc.

Recall, common challenges of Distributed Computing

1. Sending **data** to/from nodes
2. **Coordinating** among nodes
3. Recovering from node **failure**
4. Optimizing for **locality**
5. **Partition** the data to enable more parallelism

Same for all problems

How does MapReduce handle these challenges?

Fault tolerance of MapReduce

Machine failures are common in datacenters

- A MapReduce job can possibly run on thousands of machines

MapReduce simplifies fault tolerance due to the following two choices:

1. Programming model simplifies fault recovery

   e.g., **no side-effect**: a map or a reduce can simply re-execute the computation to recover from failures

   A DSM must ensure all the memory is fault tolerant

2. Builds on **a reliable service** (i.e., GFS)

Fault tolerance of MapReduce

Worker failure

- Master pings each worker periodically via heartbeat
- If no response is received within a certain time (**timeout**), the worker is marked as failed
- Map or reduce tasks given to this worker are reset back to the initial state and rescheduled for other worker (**re-execution**)
- Robust: lost 1,600 of 1,800 machines once, but finished fine

Fault tolerance of MapReduce

Master failure

- Master's state is persisted to **GFS**
- Recover master from GFS and continue

For each mapper & reduce

- Executing **state**: idle, in-progress or completed
- Locations of intermediate files

The state is checkpointed to GFS for fault tolerance

- Yet, master is unlikely to fail, since there is only one!

Fault tolerance of MapReduce

Skipping Bad Records

- Map/Reduce functions sometimes fail for some inputs
- Best solution is to debug & fix, but not always possible
- On segmentation fault:
  - Send UDP packet to master from signal handler
  - Include sequence number of record being processed
- If master sees two failures for same record:
  - Next worker is told to *skip* the record

Effect: Can work around bugs in third-party libraries

Optimization for Locality

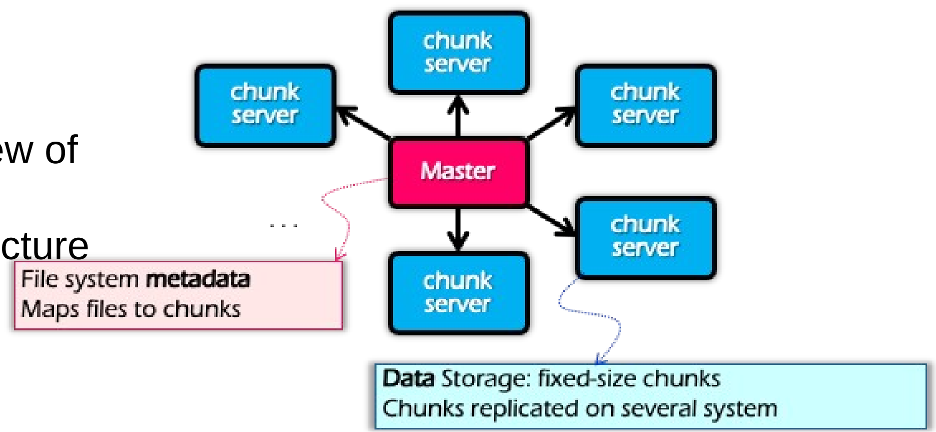
Problem: the bandwidth of datacenter network (at that time) is scarce

- If a Map worker reads all the data from the network, it would be slow

Google: Input and Output files are stored on GFS

- Google File System (SOSP'03), see previous lectures
- Each chunk is replicated on multiple servers (3)

A review of GFS architecture



Optimization for Locality

Problem: the bandwidth of datacenter network (at that time) is scarce

- If Map worker reads all the data from the network, it would be slow

Google: Input and Output files are stored on GFS

- Google File System (SOSP'03), see previous lectures
- Each chunk is replicated on multiple servers (3)

MapReduce runs on **GFS chunkservers**

- Collocate computation and storage

Master tries to **schedule** map workers on one of the nodes that has a copy of the input chunk it needs

Refinement: redundant execution

Some workers can be slower than others, aka, *stragglers*

- Other jobs consuming resources on machine
- Bad disks with soft errors transfer data slowly
- Weird things: processor caches disabled

Near end of phase, MapReduce spawn **backup copies of tasks**

- Whichever one finishes **first** “wins”
- Dramatically shortens job completion time:“*significantly reduces the time to complete large MapReduce operations* ”(check the paper)

Summary of MapReduce

Get a lot of data from **input** files

Map

- Parse & extract items of interest

Partition & Sort

Reduce

- Aggregate results

Write results to **output** files

<u>All the other issues, scheduling, fault-tolerance, data partitioning, are handled by the framework</u>

Summary of MapReduce

The user does not need to care ~~concurrency~~, ~~fault-tolerant~~, ~~data transfer~~, etc.

Reduce the user code burden and quality:

- Sort: only needs 50 LoC
- Indexing: reduce the code lines from 3800 LoC to 700LoC

Summary of MapReduce

Pros:

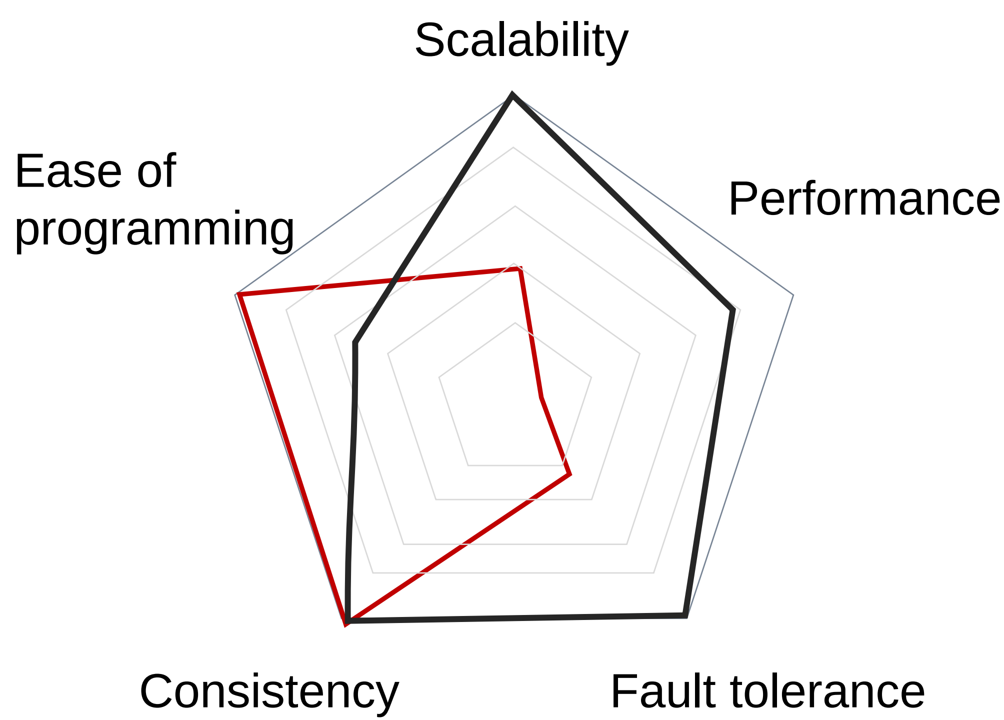
- Easy to scale
- Fault tolerant
- Good performance (depends!)
    - Good for tasks that suits MapReduce, e.g., wordcount

Cons

- Limited programming abstraction

MapReduce

Single device computing



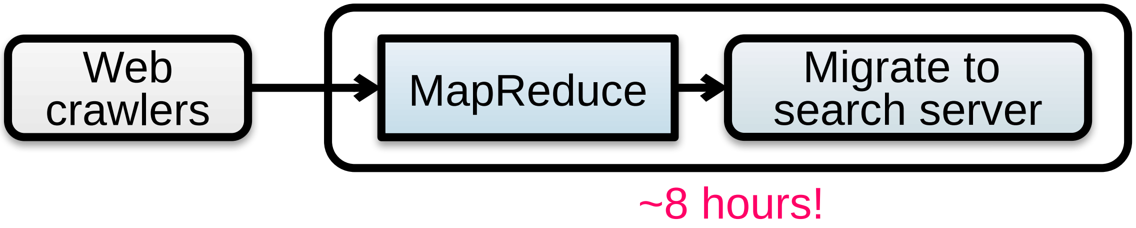
MapReduce cannot address all the issues

MapReduce was used to process webpage data collected by Google's crawlers

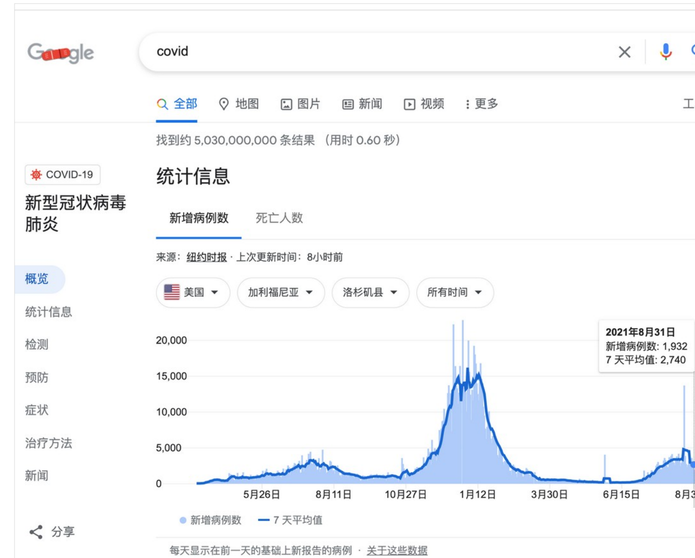
- It would extract the links and metadata needed to search the pages
- Determine the site's PageRank

The process took around **eight** hours

- Results were moved to search servers
- This was done **continuously**



MapReduce cannot address all the issues



Web has become more **dynamic**

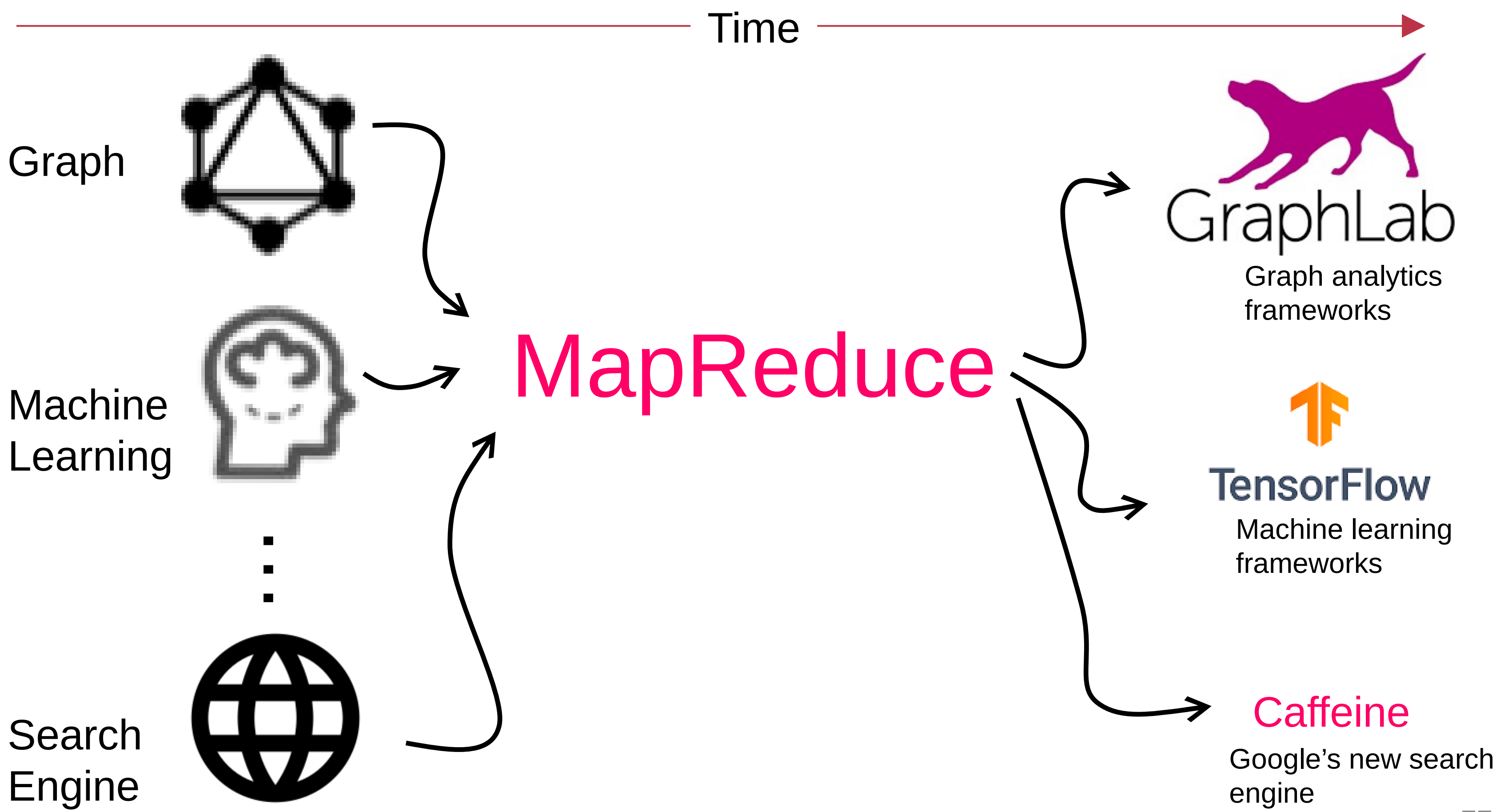
- An 8+ hour delay is a lot for some sites

Goal: refresh certain pages within seconds

MapReduce

- Batch-oriented
- Huge performance overhead
- Not suited for **near-real-time** processes
- Cannot start a new phase util previous completed
- Not optimized for specific tasks (e.g., Graph, ML)

MapReduce cannot address all the issues



Restrictiveness of MapReduce Programming Model

Question#1

– How can we use MapReduce to implement “find the five most popular pages”?

Hint:

– We can chain multiple MapReduce tasks together

```
cat /var/log/nginx/access.log | ❶
  awk '{print $7}'  | ❷
  sort              | ❸
  uniq -c           | ❹
  sort -r -n        | ❺
  head -n 5           ❻
```

Restrictiveness of MapReduce Programming Model

Question#2:

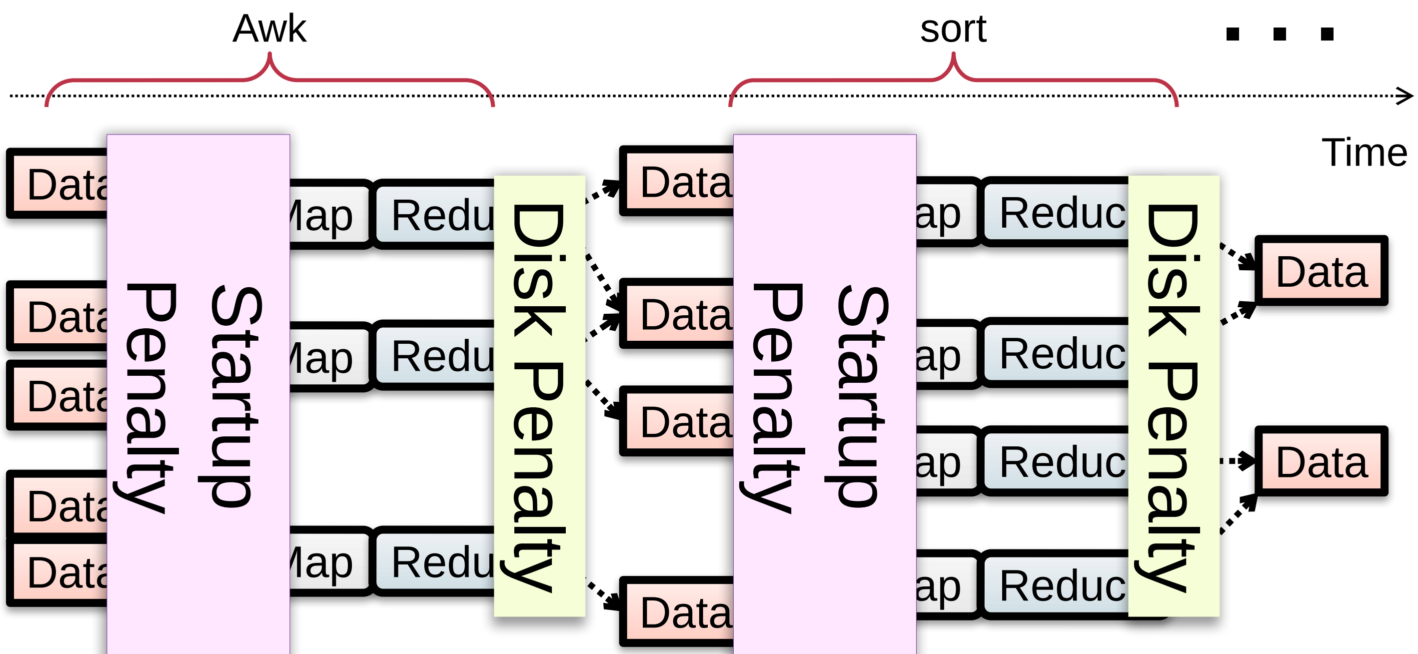
– Is chaining multiple MapReduce tasks a good solution?
  - E.g., programming is not easy
  - Fault tolerance of multiple map-reduce tasks is not supported, should be handled by the users

```
cat /var/log/nginx/access.log | ❶
  awk '{print $7}' | ❷
  sort             | ❸
  uniq -c          | ❹
  sort -r -n       | ❺
  head -n 5          ❻
```

Performance issues of multiple-sages execution

MapReduce runtime is not optimized for **iteration**

– Persistent I/O (e.g., GFS)

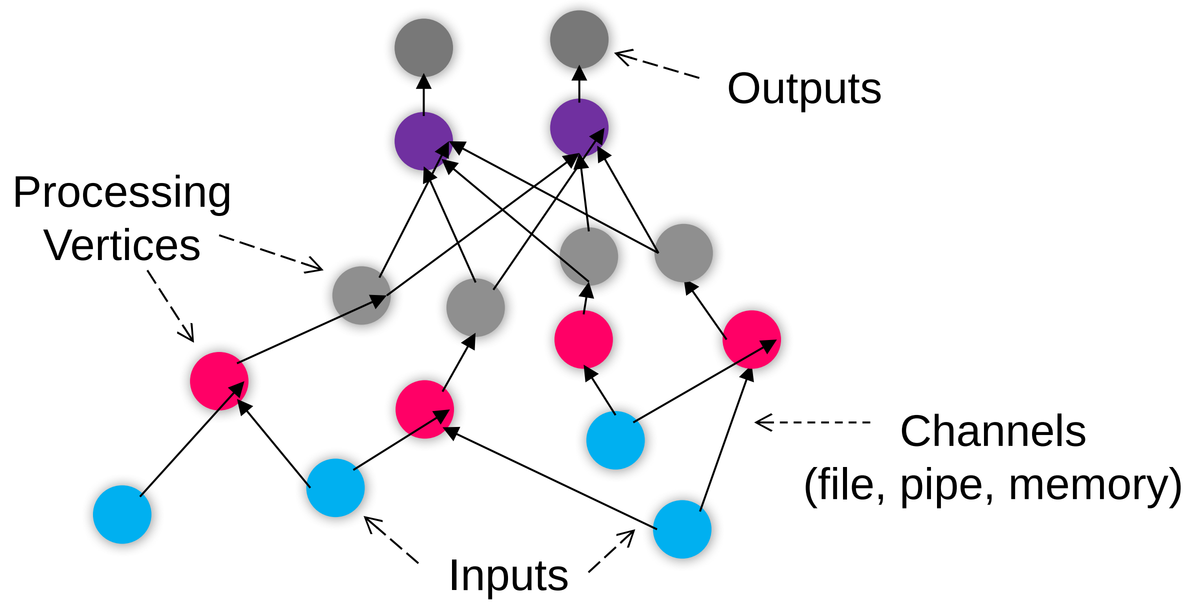


From MapReduce to DAG (Computation graph)

The computation graph abstraction

Computations are expressed as a graph (Directly acrylic graph)

- Vertices are computations
- Edges are communication channels
- Each vertex has several input and output edges

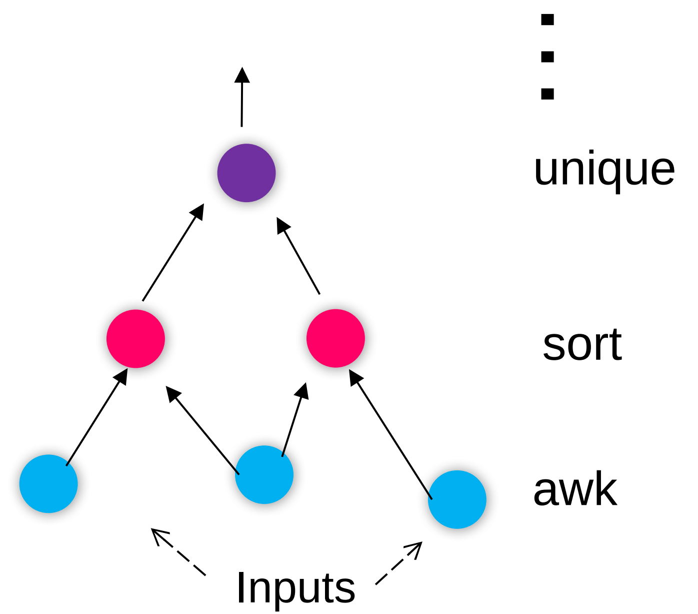


Dataflow graph can support a wide-range of jobs

Distributed computing Job = Directed Acyclic Graph (DAG)

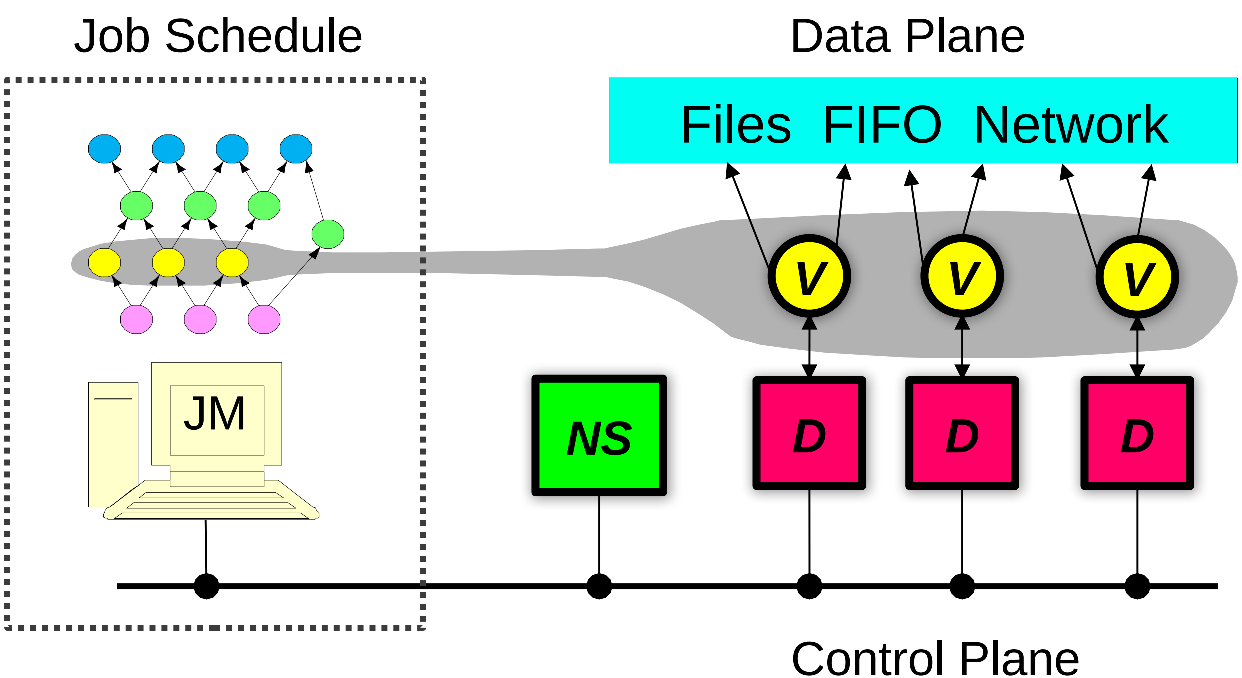
Back to our log analysis example, we can express the program as a graph

```
cat /var/log/nginx/access.log |  ❶
  awk '{print $7}' |  ❷
  sort             |  ❸
  uniq -c          |  ❹
  sort -r -n       |  ❺
  head -n 5           ❻
```



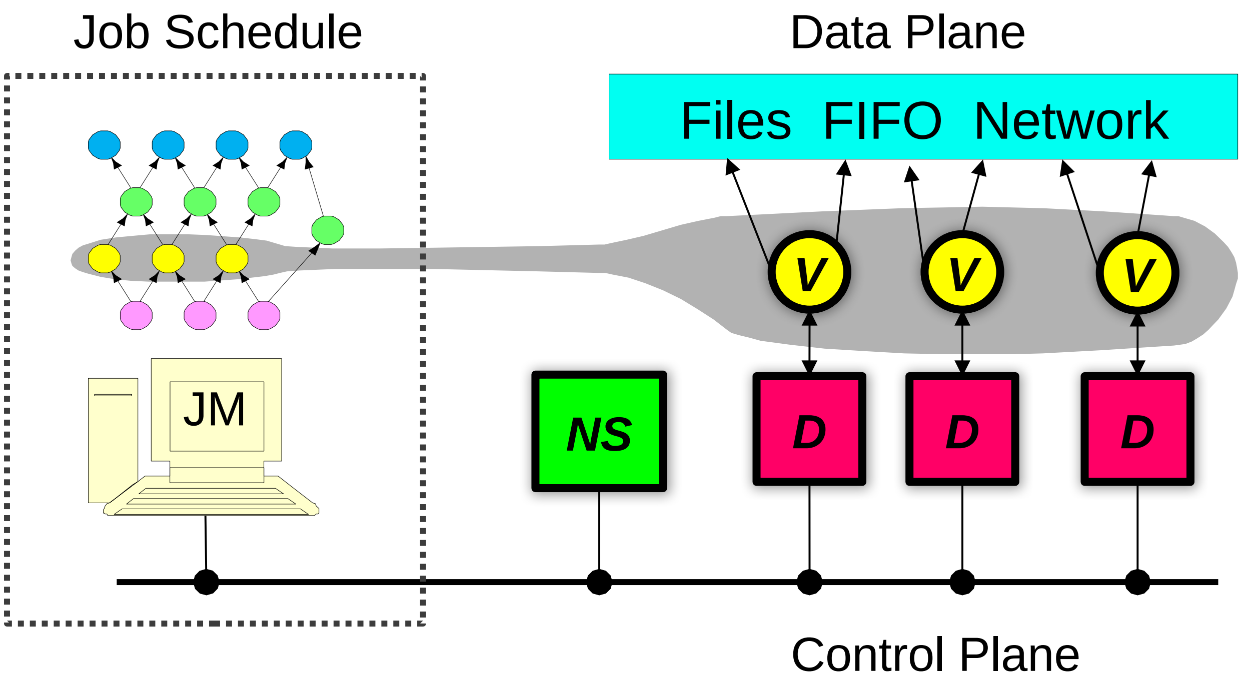
A typical DAG runtime

Vertices (V) run arbitrary app code, exchange data through TCP pipes etc., and communicate with JM to report status



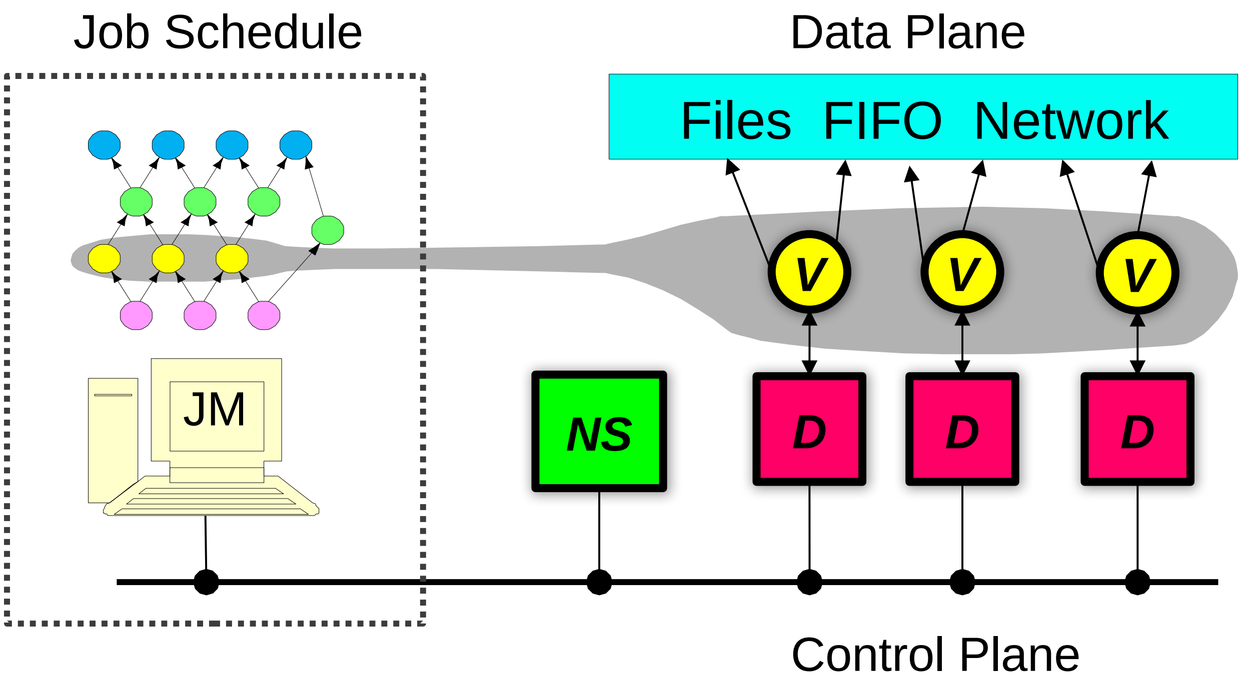
A typical DAG runtime

Job Manager (JM) consults name server (NS) to discover available nodes, and maintains job graph and schedules vertices



A typical DAG runtime

Daemon process (D) executes vertices



Scheduling in job manager

General scheduling rules

- Vertex can run anywhere once all its inputs are ready
  - Prefer executing a vertex near its inputs **(locality)**

Fault tolerance

- Vertex **fails** ✉ run it again
- Vertex's inputs are gone ✉ run upstream vertices again **(recursively)**
- Vertex is **slow** ✉ run another copy elsewhere and use output from whichever finishes **first**

What if the vertex execution is **non-idempotent** ?

Dryad proposes the DAG as distributed computing abstraction

Created by Microsoft (EuroSys'07)

- Authors: Michael Isard, Andrew Birrell, et al.

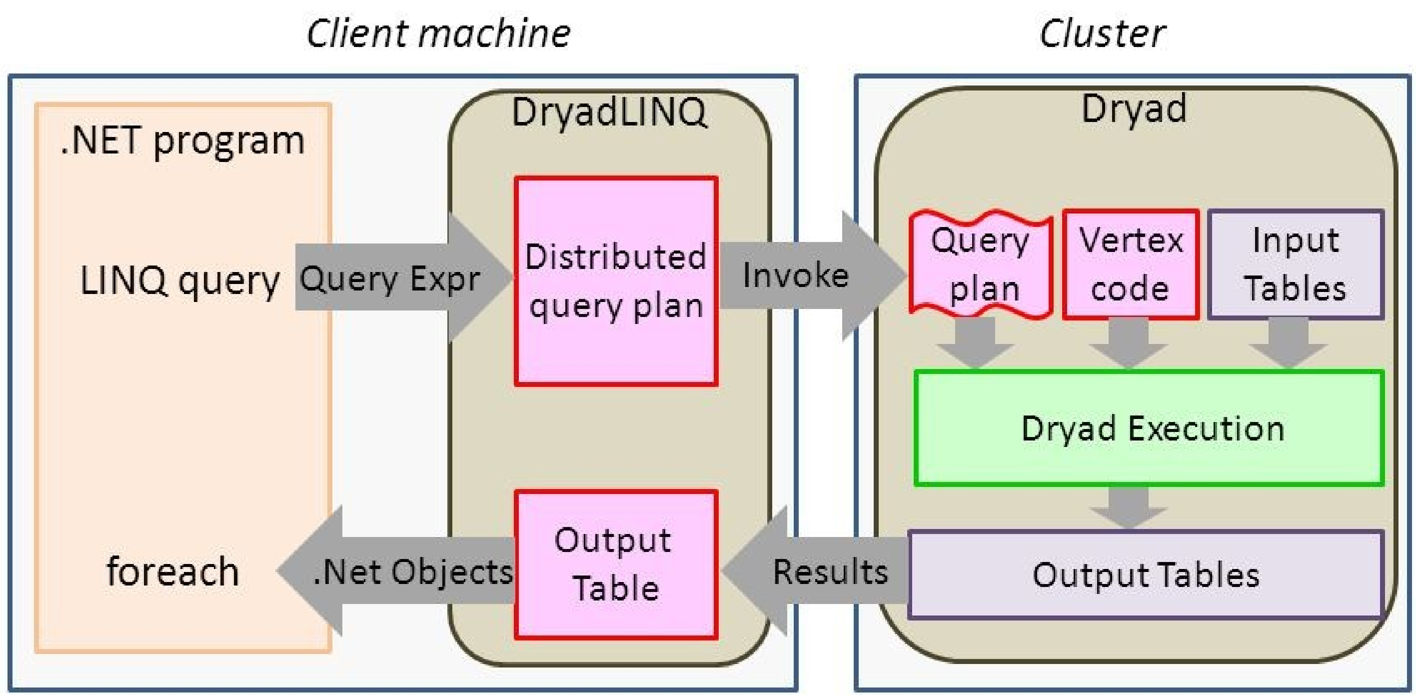
Similar goals as MapReduce

- A **general-purpose** distributed execution engine for coarse-grain data-parallel applications
- Focus on throughput, not latency
- Automatic management of scheduling, distribution, and fault tolerance

But needs application-specific semantic to split the nodes

- Otherwise, the graph fallback to a chain, so there is no parallelism
- A little harder than MapReduce, but also hides the distributed execution details

DryadLINQ: Dryad + LINQ (SQL like)



More friendly to traditional data programmers: use LINQ (SQL-like language)

Summary

> Dryad lets developers easily create large-scale distributed apps without requiring them to master any concurrency techniques beyond being able to draw a graph of the data dependencies of their algorithms.
>
> *-- Michael Isard*

Sacrifice some architectural **simplicity** compared with MapReduce system design

Provide more **flexible** abstract to developers expressing their code as a **DAG**