Algorithms for MapReduce

Admin

Homework 1 and Lab 1 on website
Labs start tomorrow

241 enrolled

Takeaways

Design MapReduce computations in pseudocode

Optimize a computation, with motivation

Patterns used

Less Important

These specific examples

Problem: Comparing Output

Alice's Word Counts

a	20
hi	2

i	13
the	31

why	12

Bob's Word Counts

a	20
why	12

hi	2
i	13
the	31

Problem: Comparing Output

Alice's Word Counts

a	20
hi	2

i	13
the	31

why	12

Bob's Word Counts

a	20
why	12

hi	2
i	13
the	31

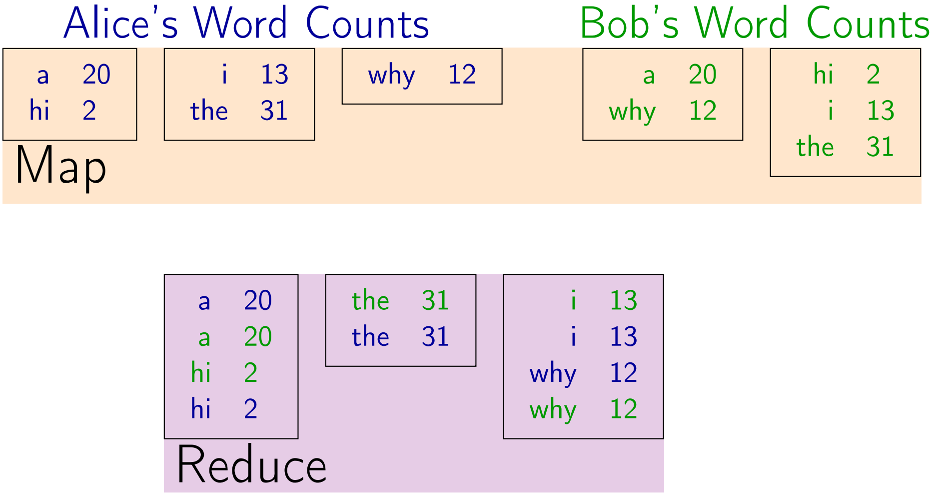
a	20
a	20
hi	2
hi	2

the	31
the	31

i	13
i	13
why	12
why	12

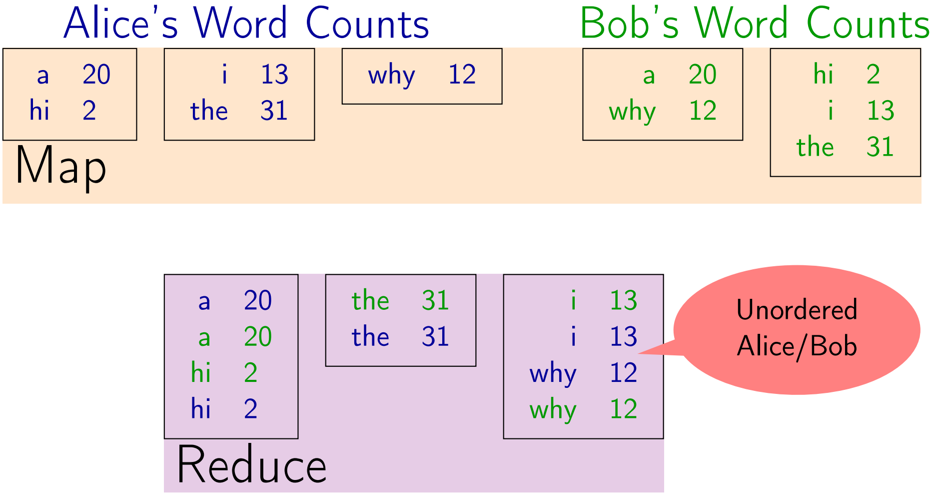
Send words to a consistent place

Problem: Comparing Output



Send words to a consistent place: reducers

Problem: Comparing Output



Send words to a consistent place: reducers

Comparing Output Detail

Map: (word, count) $\mapsto$ (word, student, count) [1]

Reduce: Verify both values are present and match.
Deduct marks from Alice/Bob as appropriate.

[1] The mapper can tell Alice and Bob apart by input file name.

Comparing Output Detail

Map: (word, count) $\mapsto$ (word, student, count) [1]

Partition: By word

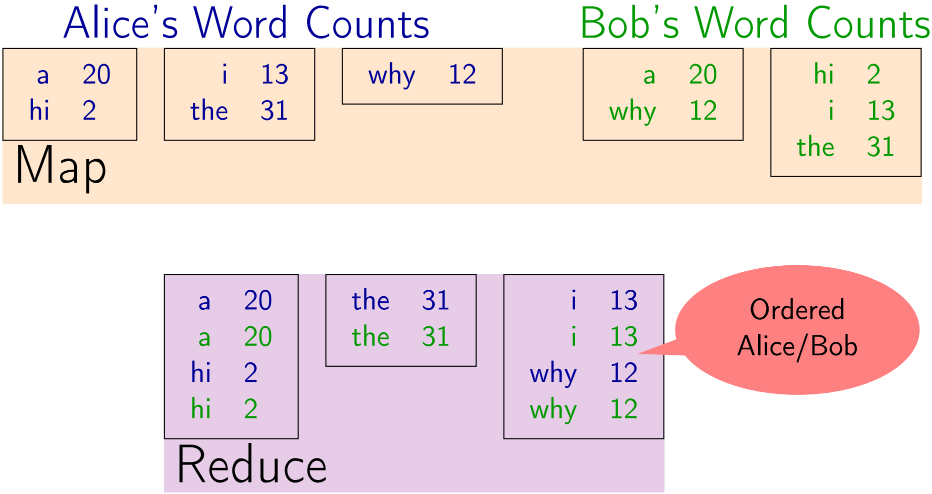
Sort: By ~~word~~(word, student)

Reduce: Verify both values are present and match.
Deduct marks from Alice/Bob as appropriate.

Exploit sort to control input order

[1] The mapper can tell Alice and Bob apart by input file name.

Problem: Comparing Output



Send words to a consistent place: reducers

Pattern: Exploit the Sort

Without Custom Sort

Reducer buffers all students in RAM

$\Longrightarrow$

Might run out of RAM

With Custom Sort

TA appears first, reducer streams through students.

Constant reducer memory.

We will give higher marks to scalable solutions
(even if yours runs on small data)

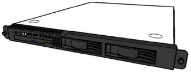
THE UNIVERSITY *of* EDINBURGH
informatics

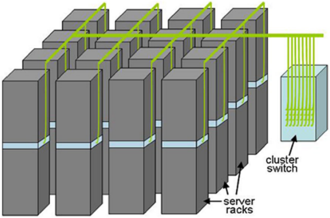
PROGRAMMING FOR A DATA CENTRE

Programming for a data centre

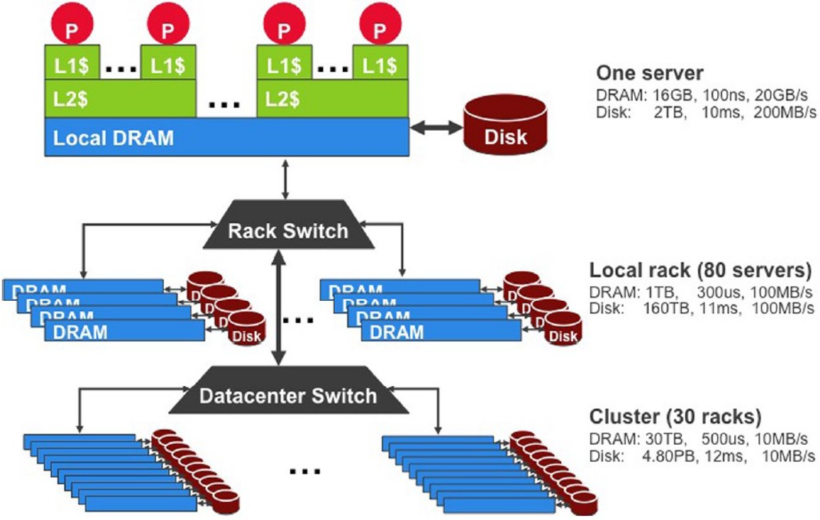
- Understanding the design of warehouse-sized computes
  - Different techniques for a different setting
  - Requires quite a bit of rethinking
- MapReduce algorithm design
  - How do you express everything in terms of `map()`, `reduce()`, `combine()`, and `partition()`?
  - Are there any design patterns we can leverage?

Building Blocks





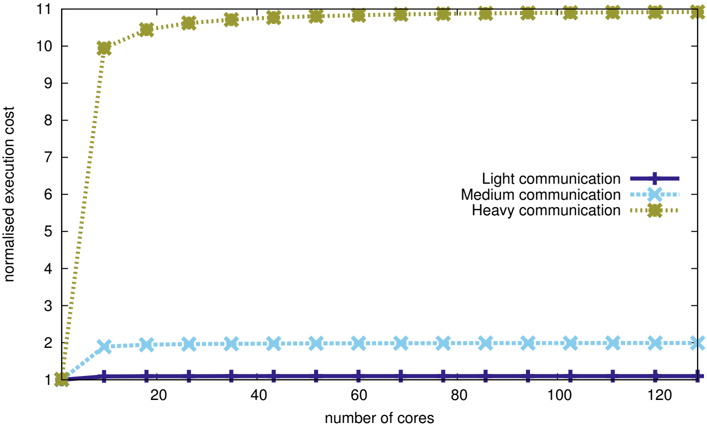
Storage Hierarchy



Scaling up vs. out

- No single machine is large enough
  - Smaller cluster of large SMP machines vs. larger cluster of commodity machines (e.g., 8 128-core machines vs. 128 8-core machines)
- Nodes need to talk to each other!
  - Intra-node latencies: ~100 ns
  - Inter-node latencies: ~100 μs

Overhead of communication



Seeks vs. scans

- Consider a 1TB database with 100 byte records
  - We want to update 1 percent of the records
- Scenario 1: random access
  - Each update takes ~30 ms (seek, read, write)
  - 10^8 updates = ~35 days
- Scenario 2: rewrite all records
  - Assume 100MB/s throughput
  - Time = 5.6 hours(!)
- Lesson: avoid random seeks!

Numbers everyone should know

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA → Netherlands → CA	150,000,000 ns

THE UNIVERSITY of EDINBURGH
informatics

DEVELOPING ALGORITHMS

Optimising computation

- The cluster management software orchestrates the computation
- But we can still optimise the computation
  - Just as we can write better code and use better algorithms and data structures
  - At all times confined within the capabilities of the framework
- Cleverly-constructed data structures
  - Bring partial results together
- Sort order of intermediate keys
  - Control order in which reducers process keys
- Partitioner
  - Control which reducer processes which keys
- Preserving state in mappers and reducers
  - Capture dependencies across multiple keys and values

Importance of local aggregation

- Ideal scaling characteristics:
  - Twice the data, twice the running time
  - Twice the resources, half the running time
- Why can't we achieve this?
  - Synchronization requires communication
  - Communication kills performance
- Thus… avoid communication!
  - Reduce intermediate data via local aggregation
  - Combiners can help

Word count: baseline

```
class Mapper
  method map(docid a, doc d)
    for all term t in d do
      emit(t, 1);

class Reducer
  method reduce(term t, counts [c1, c2, …])
    sum = 0;
    for all counts c in [c1, c2, …] do
      sum = sum + c;
    emit(t, sum);
```

Word count: introducing combiners

```
class Mapper
  method map(docid a, doc d)
    H = associative_array(term → count;)
    for all term t in d do
      H[t]++;
    for all term t in H[t] do
      emit(t, H[t]);
```

Local aggregation reduces further computation

Word count: introducing combiners

```
class Mapper
  method initialise()
    H = associative_array(term → count);

  method map(docid a, doc d)
    for all term t in d do
      H[t]++;

  method close()
    for all term t in H[t] do
      emit(t, H[t]);
```

Compute sums *across* documents!

THE UNIVERSITY of EDINBURGH
informatics

Design pattern for local aggregation

- In-mapper combining
  - Fold the functionality of the combiner into the mapper by preserving state across multiple map calls
- Advantages
  - Speed
  - Why is this faster than actual combiners?
- Disadvantages
  - Explicit memory management required
  - Potential for order-dependent bugs

Problem: Averaging

We're given temperature readings from cities:

Key	**Value**
San Francisco	22
Edinburgh	14
Los Angeles	23
Edinburgh	12
Edinburgh	9
Los Angeles	21

Find the average temperature in each city.

Map: (city, temperature) $\mapsto$ (city, temperature)

Reduce: Count, sum temperatures, and divide.

Problem: Averaging

We're given temperature readings from cities:

Key	**Value**
San Francisco	22
Edinburgh	14
Los Angeles	23
Edinburgh	12
Edinburgh	9
Los Angeles	21

Find the average temperature in each city.

Map: (city, temperature) $\mapsto$ (city, temperature)
Combine: Same as reducer?
Reduce: Count, sum temperatures, and divide.

Problem: Averaging

We're given temperature readings from cities:

Key	**Value**
San Francisco	22
Edinburgh	14
Los Angeles	23
Edinburgh	12
Edinburgh	9
Los Angeles	21

Find the average temperature in each city.

Map: (city, temperature) $\mapsto$ (city, count $= 1$, temperature)
Combine: Sum count and temperature fields.
Reduce: Sum count, sum temperatures, and divide.

Pattern: Combiners

Combiners reduce communication by aggregating locally.

Many times they are the same as reducers (i.e. summing).

…but not always (i.e. averaging).

Algorithm design: term co-occurrence

- Term co-occurrence matrix for a text collection
  - M = N x N matrix (N = vocabulary size)
  - M_{ij}: number of times *i* and *j* co-occur in some context (for concreteness, let's say context = sentence)
- Why?
  - Distributional profiles as a way of measuring semantic distance
  - Semantic distance useful for many language processing tasks

Using MapReduce for large counting problems

- Term co-occurrence matrix for a text collection is a specific instance of a large counting problem
  - A large event space (number of terms)
  - A large number of observations (the collection itself)
  - Goal: keep track of interesting statistics about the events
- Basic approach
  - Mappers generate partial counts
  - Reducers aggregate partial counts

How do we aggregate partial counts efficiently?

First try: pairs

- Each mapper takes a sentence:
  - Generate all co-occurring term pairs
  - For all pairs, emit (a, b) → count
- Reducers sum up counts associated with these pairs
- Use combiners!

Pairs: pseudo-code

```
class Mapper
  method map(docid a, doc d)
    for all w in d do
      for all u in neighbours(w) do
        emit(pair(w, u), 1);

class Reducer
  method reduce(pair p, counts [c1, c2, …])
    sum = 0;
    for all c in [c1, c2, …] do
      sum = sum + c;
    emit(p, sum);
```

Analysing pairs

- Advantages
  - Easy to implement, easy to understand
- Disadvantages
  - Lots of pairs to sort and shuffle around (upper bound?)
  - Not many opportunities for combiners to work

Another try: stripes

- Idea: group together pairs into an associative array

```
(a, b) → 1
(a, c) → 2
(a, d) → 5          a → { b: 1, c: 2, d: 5, e: 3, f: 2 }
(a, e) → 3
(a, f) → 2
```

- Each mapper takes a sentence:
  – Generate all co-occurring term pairs
  – For each term, emit a → { b: count_b, c: count_c, d: count_d … }
- Reducers perform element-wise sum of associative arrays

```
a → { b: 1,       d: 5, e: 3 }
a → { b: 1, c: 2, d: 2,       f: 2 }
a → { b: 2, c: 2, d: 7, e: 3, f: 2 }
```

Cleverly-constructed data structure brings together partial results

Stripes: pseudo-code

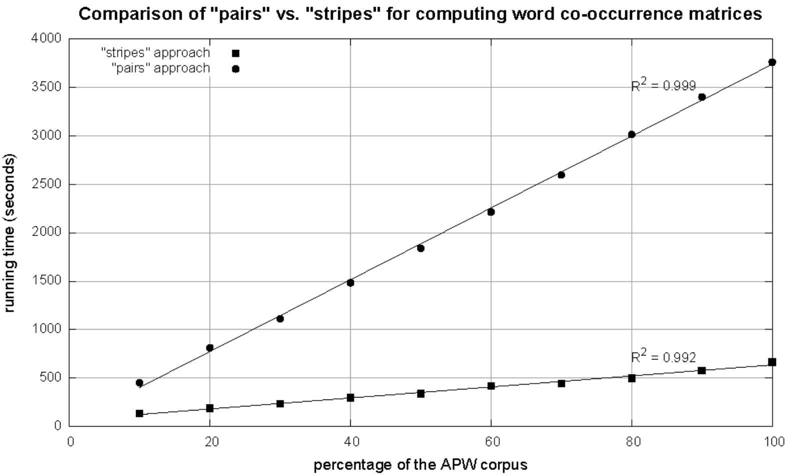
```
class Mapper
  method map(docid a, doc d)
    for all w in d do
      H = associative_array(string → integer);
      for all u in neighbours(w) do
        H[u]++;
      emit(w, H);

class Reducer
  method reduce(term w, stripes [H1, H2, …])
    Hf = assoiative_array(string → integer);
    for all H in [H1, H2, …] do
      sum(Hf, H);    // sum same-keyed entries
    emit(w, Hf);
```

Stripes analysis

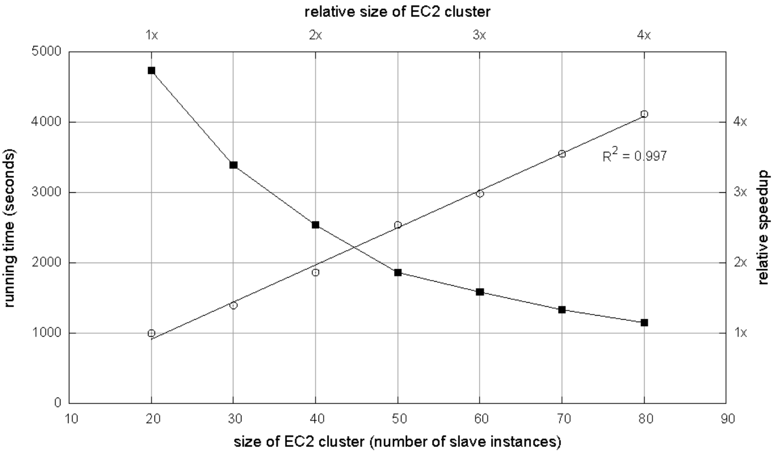
- Advantages
  - Far less sorting and shuffling of key-value pairs
  - Can make better use of combiners
- Disadvantages
  - More difficult to implement
  - Underlying object more heavyweight
  - Fundamental limitation in terms of size of event space

THE UNIVERSITY of EDINBURGH
informatics



Cluster size: 38 cores
Data Source: Associated Press Worldstream (APW) of the English Gigaword Corpus (v3), which contains 2.27 million documents (1.8 GB compressed, 5.7 GB uncompressed)

Effect of cluster size on "stripes" algorithm



Debugging at scale

- Works on small datasets, won't scale… why?
  - Memory management issues (buffering and object creation)
  - Too much intermediate data
  - Mangled input records
- Real-world data is messy!
  - There's no such thing as consistent data
  - Watch out for corner cases
  - Isolate unexpected behavior, bring local

Summary

- Further delved into computing using MapReduce
- Introduced map-side optimisations
- Discussed why certain things may not work as expected
- Need to be really careful when designing algorithms to deploy over large datasets
- What seems to work on paper may not be correct when distribution/ parallelisation kick in