MapReduce Systems

Sara Bouchenak

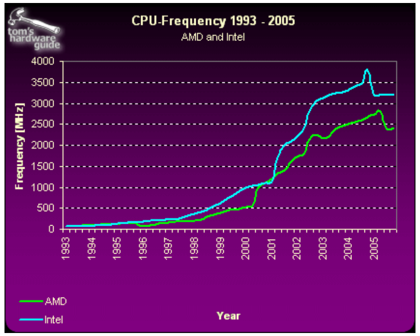
Sara.Bouchenak@imag.fr
http://sardes.inrialpes.fr/~bouchena/teaching/

- Lectures based on the following slides:
  - http://code.google.com/edu/submissions/mapreduce-minilecture/listing.html
- Authors:
  - Christophe Bisciglia, Aaron Kimball, Sierra Michels-Slettvet

Outline

- Part I: Motivations
  - Introduction
  - Parallel vs. Distributed Computing
  - History of Distributed Computing
  - Parallelization and Synchronization
- Part II: MapReduce theory and implementation
  - Lisp/ML review (functional programming, map, fold)
  - MapReduce overview
  - Hadoop

Computer Speedup



Moore's Law: "*The density of transistors on a chip doubles every 18 months, for the same cost*" (1965)

Scope of problems

- What can you do with 1 computer?
- What can you do with 100 computers?
- What can you do with an entire data center?

Distributed problems

- Rendering multiple frames of high-quality animation

Image: DreamWorks Animation and not subject to the Creative Commons license applicable to the rest of this work.

Distributed problems

- Simulating several hundred or thousand characters

Happy Feet © Kingdom Feature Productions;
Lord of the Rings © New Line Cinema, neither image is subject to the Creative Commons license applicable to the rest of the work.

Distributed problems

- Indexing the web (Google)
- Simulating an Internet-sized network for networking experiments (PlanetLab)
- Speeding up content delivery (Akamai)

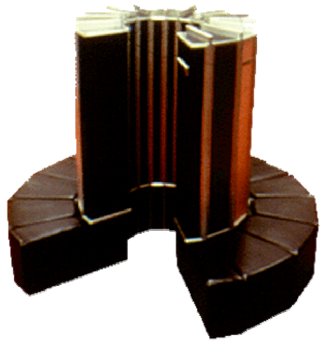
What is the key attribute that all these examples have in common?

Parallel vs. Distributed

- Parallel computing can mean:
  - Vector processing of data
  - Multiple CPUs in a single computer
- Distributed computing is multiple CPUs across many computers over the network

A Brief History… 1975-85

- Parallel computing was favored in the early years
- Gradually more thread-based parallelism was introduced



A Brief History… 1985-95

- "Massively parallel architectures" start rising in prominence
- Message Passing Interface (MPI) and other libraries developed
- Bandwidth was a big problem

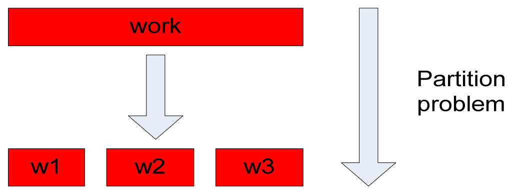
A Brief History… 1995-Today

- Cluster/grid architecture increasingly dominant
- Special node machines eschewed in favor of COTS technologies
- Web-wide cluster software
- Companies like Google take this to the extreme

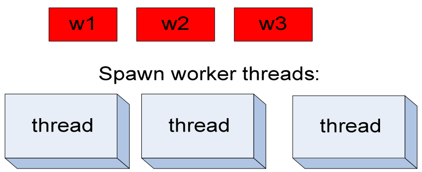
Parallelization & Synchronization

Parallelization Idea

- Parallelization is “easy” if processing can be cleanly split into n units:

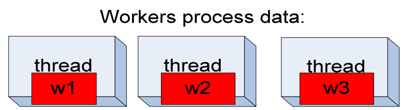


Parallelization Idea (2)

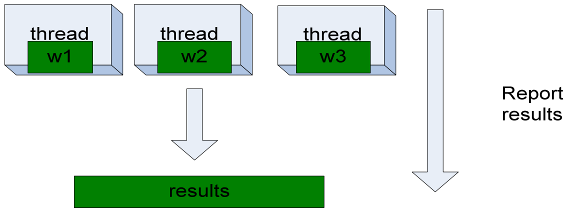


In a parallel computation, we would like to have as many threads as we have processors. e.g., a four-processor computer would be able to run four threads at the same time.

Parallelization Idea (3)



Parallelization Idea (4)



Parallelization Pitfalls

But this model is too simple!

- How do we assign work units to worker threads?
- What if we have more work units than threads?
- How do we aggregate the results at the end?
- How do we know all the workers have finished?
- What if the work cannot be divided into completely separate tasks?

What is the common theme of all of these problems?

Parallelization Pitfalls (2)

- Each of these problems represents a point at which multiple threads must communicate with one another, or access a shared resource.
- Golden rule: Any memory that can be used by multiple threads must have an associated *synchronization system*!

What is Wrong With This?

Thread 1:

```
void foo() {
  x++;
  y = x;
}
```

Thread 2:

```
void bar() {
  y++;
  x+=3;
}
```

If the initial state is y = 0, x = 6, what happens after these threads finish running?

Multithreaded = Unpredictability

- Many things that look like "one step" operations actually take several steps under the hood:

Thread 1:
```
void foo() {
  eax = mem[x];
  inc eax;
  mem[x] = eax;
  ebx = mem[x];
  mem[y] = ebx;
}
```

Thread 2:
```
void bar() {
  eax = mem[y];
  inc eax;
  mem[y] = eax;
  eax = mem[x];
  add eax, 3;
  mem[x] = eax;
}
```

- When we run a multithreaded program, we don't know what order threads run in, nor do we know when they will interrupt one another.

Multithreaded = Unpredictability

This applies to more than just integers:

- Pulling work units from a queue
- Reporting work back to master unit
- Telling another thread that it can begin the "next phase" of processing

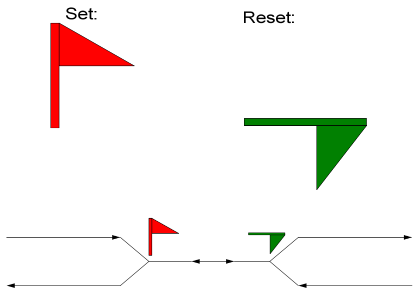
… All require synchronization!

Synchronization Primitives

- A *synchronization primitive* is a special shared variable that guarantees that it can only be accessed **atomically**.
- Hardware support guarantees that operations on synchronization primitives only ever take one step

Semaphores

- A semaphore is a flag that can be raised or lowered in one step
- Semaphores were flags that railroad engineers would use when entering a shared track



Only one side of the semaphore can ever be red! (Can both be green?)

Semaphores

- set() and reset() can be thought of as lock() and unlock()
- Calls to lock() when the semaphore is already locked cause the thread to **block**.
- *Pitfalls: Must "bind" semaphores to particular objects; must remember to unlock correctly*

The "corrected" example

Thread 1:

```
void foo() {
  sem.lock();
  x++;
  y = x;
  sem.unlock();
}
```

Thread 2:

```
void bar() {
  sem.lock();
  y++;
  x+=3;
  sem.unlock();
}
```

Global var "Semaphore sem = new Semaphore();" guards access to x & y

Condition Variables

- A condition variable notifies threads that a particular condition has been met
- Inform another thread that a queue now contains elements to pull from (or that it's empty – request more elements!)
- *Pitfall: What if nobody's listening?*

The final example

Thread 1:

```
void foo() {
  sem.lock();
  x++;
  y = x;
  fooDone = true;
  sem.unlock();
  fooFinishedCV.notify();
}
```

Thread 2:

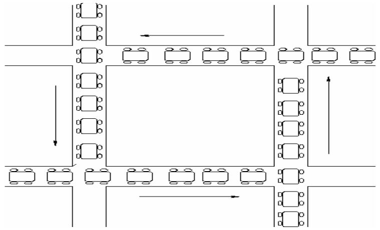
```
void bar() {
  sem.lock();
  if(!fooDone)
    fooFinishedCV.wait(sem);
  y++;
  x+=3;
  sem.unlock();
}
```

Global vars: Semaphore sem = new Semaphore(); ConditionVar fooFinishedCV = new ConditionVar(); boolean fooDone = false;

Too Much Synchronization? Deadlock

Synchronization becomes even more complicated when multiple locks can be used

Can cause entire system to "get stuck"



Thread A:
```
semaphore1.lock();
semaphore2.lock();
/* use data guarded by
      semaphores */
semaphore1.unlock();
semaphore2.unlock();
```

Thread B:
```
semaphore2.lock();
semaphore1.lock();
/* use data guarded by
      semaphores */
semaphore1.unlock();
semaphore2.unlock();
```

(Image: RPI CSCI.4210 Operating Systems notes)

The Moral: Be Careful!

- Synchronization is hard
  - Need to consider all possible shared state
  - Must keep locks organized and use them consistently and correctly
- Knowing there are bugs may be tricky; fixing them can be even worse!
- Keeping shared state to a minimum reduces total system complexity

Outline

- *Part I: Motivations*
  - *Introduction*
  - *Parallel vs. Distributed Computing*
  - *History of Distributed Computing*
  - *Parallelization and Synchronization*
- **Part II: MapReduce theory and implementation**
  - Lisp/ML review (functional programming, map, fold)
  - MapReduce overview
  - Hadoop

Functional Programming Review

- Functional operations do not modify data structures: They always create new ones
- Original data still exists in unmodified form
- Data flows are implicit in program design
- Order of operations does not matter

Functional Programming Review

fun foo(l: int list) =
  sum(l) + mul(l) + length(l)

Order of sum() and mul(), etc does not matter – they do not modify *l*

Functional Updates Do Not Modify Structures

fun append(x, lst) =
  let lst' = reverse lst in
    reverse (x :: lst')

The append() function above reverses a list, adds a new element to the front, and returns all of that, reversed, which appends an item.

But it *never modifies lst*!

Functions Can Be Used As Arguments

fun DoDouble(f, x) = f (f x)

It does not matter what f does to its argument; DoDouble() will do it twice.

MapReduce

Motivation: Large Scale Data Processing

- Want to process lots of data (> 1 TB)
- Want to parallelize across hundreds/thousands of CPUs
- … Want to make this easy

MapReduce

- Automatic parallelization & distribution
- Fault-tolerant
- Provides status and monitoring tools
- Clean abstraction for programmers

Programming Model

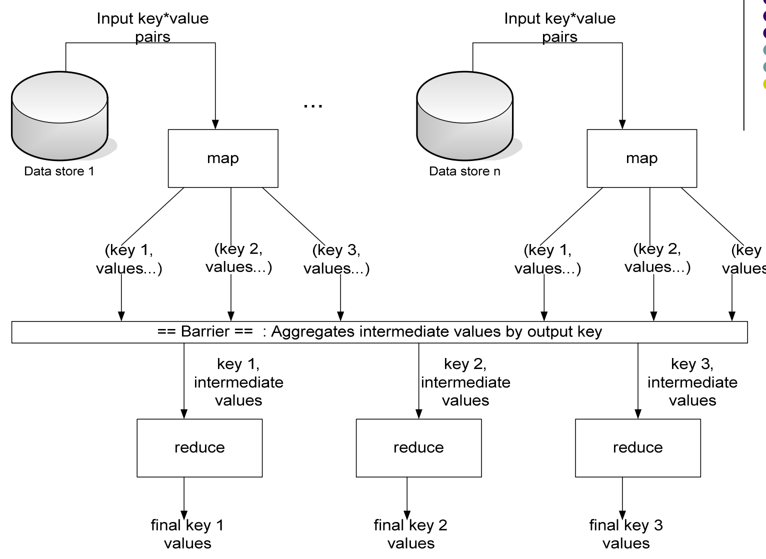
- Borrows from functional programming
- Users implement interface of two functions:
  - `map (in_key, in_value) -> (out_key, intermediate_value) list`
  - `reduce (out_key, intermediate_value list) -> out_value list`

map

- Records from the data source (lines out of files, rows of a database, etc) are fed into the map function as key*value pairs: e.g., (filename, line).
- map() produces one or more *intermediate* values along with an output key from the input.

reduce

- After the map phase is over, all the intermediate values for a given output key are combined together into a list
- reduce() combines those intermediate values into one or more *final values* for that same output key
- (in practice, usually only one final value per key)



Parallelism

- map() functions run in parallel, creating different intermediate values from different input data sets
- reduce() functions also run in parallel, each working on a different output key
- All values are processed *independently*
- Bottleneck: reduce phase can't start until map phase is completely finished.

Example: Count word occurrences

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

reduce(String output_key, Iterator
  intermediate_values):
  // output_key: a word
  // output_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v);
 Emit(AsString(result));
```

Example vs. Actual Source Code

- Example is written in pseudo-code
- Actual implementation is in C++, using a MapReduce library
- Bindings for Python and Java exist via interfaces
- True code is somewhat more involved (defines how the input key/values are divided up and accessed, etc.)

Locality

- Master program divvies up tasks based on location of data: tries to have map() tasks on same machine as physical file data, or at least same rack
- map() task inputs are divided into 64 MB blocks: same size as Google File System chunks

Fault Tolerance

- Master detects worker failures
  - Re-executes completed & in-progress map() tasks
  - Re-executes in-progress reduce() tasks
- Master notices particular input key/values cause crashes in map(), and skips those values on re-execution.
  - Effect: Can work around bugs in third-party libraries!

Optimizations

- No reduce can start until map is complete:
  - A single slow disk controller can rate-limit the whole process
- Master redundantly executes “slow-moving” map tasks; uses results of first copy to finish

Why is it safe to redundantly execute map tasks? Wouldn’t this mess up the total computation?

MapReduce Conclusions

- MapReduce has proven to be a useful abstraction
- Greatly simplifies large-scale computations at Google
- Functional programming paradigm can be applied to large-scale applications
- Fun to use: focus on problem, let library deal w/ messy details

Hadoop

- Apache Hadoop project develops open-source software for reliable, scalable, distributed computing
- MapReduce implementation
- Who uses Hadoop
  - Amazon
  - Adobe
  - Facebook
  - FOX
  - Google
  - IBM
  - LinkedIn
  - …

Outline

- *Part I: Motivations*
  - *Introduction*
  - *Parallel vs. Distributed Computing*
  - *History of Distributed Computing*
  - *Parallelization and Synchronization*
- *Part II: MapReduce theory and implementation*
  - *Lisp/ML review (functional programming, map, fold)*
  - *MapReduce overview*
  - *Hadoop*