Parallel Cluster Programming

CMU 15-440: Distributed Systems

Last Time

- Questions about Paxos / distributed consensus?
- If you haven't picked up your exam yet, it's with my admin (see web page)

Parallelism ubiquitous

- Even your laptop, if your parents bought you something decent :), has 2 or more cores in it.
  - Each of those cores is actually internally parallel, but that's not this course.
- The question of the decade in computing (not exaggerating): How to exploit parallelism.

Q1: What's the workload??

- Before we try to solve it, let's look briefly at the map of problems and solutions...
- "Trivially parallel" -- e.g., password cracking. Requires little communication, memory, etc.
- Compute-intensive -- e.g., physical mesh simulations in HPC. Often very memory intensive, need low-latency, high-bw communication
- Data-intesive -- e.g., computing an index of the web, data-mining, etc. Often disk intensive, bandwidth intensive, but not as latency sensitive.

Q2: The challenges

- First: Algorithmic. How do you solve problem X in parallel?
  - Sometimes this is very hard; sometimes it's straightforward. For our purposes, let's assume the answer is known. Otherwise, take Guy Blelloch's course.
- Second: Systems. How do you *practically* solve problem X using reasonable amounts of programmer time, efficiently use the parallel resources, etc.?

Helping Programmers Use Clusters

- As systems people, our job is to make stuff work. "Tools to make tools." What tools can we provide to ease the pain of parallel computing?
- Step 0: We can make it easy to execute code on lots of machines... and to share those machines between "jobs" (cf Condor)
  - Step 0a: We can provide you with a shared distributed filesystem, or copy your programs onto these machines automatically
- Step 1: We can give you easy to use mechanisms for communication between processes (RPC, or, in the high-performance domain, MPI.)
- Step 1a: MPI slightly higher level: library knows which other computers are involved in the processing, handles process spawning (step 0), provides message sending, broadcast, many-to-1, get(), put() abstraction for shared memory regions, synchronization, etc.)
- Observations:
  - These tools are very general - you can use them to solve any parallel problem
  - These tools are very low-level - they don't deal with a lot of very hard questions in parallel programming!

Some hard questions

- Psst - you saw a lot of these in lab 1!
- Dealing with failures (it's tough - even your professor puts bugs into complex algorithms when explaining them!)
- Dealing with load balance - efficiently keeping all of the machines busy, splitting up the work into chunks, etc.
- Parallelizing the algorithm, etc.

More helping programmers

- Provide libraries of parallel algorithms they can just use (matrix multiply, etc.)
- Provide generic failure recovery: checkpoint & restore.
  - Basic idea: Take a "snapshot" of the state of all programs on the cluster, save it to disk. If you fail, restart everyone from this checkpoint.
  - Very general, very expensive.
- Remember back to the start of the class, I mentioned that you could try to provide a generic shared memory abstraction across a cluster, but that people had mostly given up (too expensive)? Too much generality here hurts, too.
- *Are there salient features of a large class of parallel applications we can exploit to make life easier?*
- Yes, for the class of data-intensive applications.
  - Wha wha? Imagine: Count how many times dga searched for "free distributed systems lectures" on Google, from all of Google's logfiles.
  - You might imagine that Google's logfiles are very, very large.

Helping helping

- An observation: If you can express a problem either parallel functionally (no side effects, defining per-element operations), or sometimes declaratively, it's often easier to do them in parallel
- Example: SQL
  - SELECT * from profs where id='dga';
  - I could do this by, in parallel, checking every record in the database against id 'dga' and then returning a list of the matching ones
- Example: ML
  - let timestwo a = (a + a);;
  - List.map timestwo [1; 2; 3;];;   ----> int list [2; 4; 6;]

Let's look at that map more closely

- In what order do we have to apply the function to the elements?
  - A: Any one we want, even in parallel. The function has no side effects - the applications are independent
- What happens if we apply the function to the same element twice?
  - A: Nothing, it's safe to re-do it and recompute the value - no side effects! :)
- Suggests a nice basis for both parallelization and fault tolerance...
- But programmer does not live by map() alone.

Google's MapReduce

- A *model*, and the name of Google's implementation.
- Unlike the reduce (or fold, in ml) you're used to, theirs is partitioned by key.
  - That's pretty much the only difference.
- Open source implementation of the model: Hadoop (affiliated with Yahoo!)

- map transforms a set of key, value pairs to a different set of key, value pairs, operating on each input pair indepdendently:
- map f [(k1, v1), (k2, v2), (k3, v3), ...] ->
  [(kx, vx), (ky, vy), (kz, vz), (ka, vb), ...]
- f (k, v) -> [(k', v'), ...)
  - map function can output *zero or more* key, value pairs for every input item
- reduce (k, [v1, v2, ...]) -> [vN, vM, ...]
  - Usually, reduce outputs one value for each key

Example: Word count

```
// key = document name, value = document contents
map(String key, String value):
    for each word w in value:
      EmitIntermediate(w, “1”)

 reduce(String key, list values):
    int count = 0
    for each v in values:
      count += parseInteger(v)
    Emit(string(count))
```

Things to note

- The system automatically groups the results by the key used in the reduce, so the output of that would be:
  - (frog, 5), (dog, 100), (systems, 1000), (15-213, 5)

- A “reduce task” runs on a machine. It may handle multiple keys for reducing.
- (k, v) pairs are allocated to reduce tasks based on the key (“partitioning”), so that one reduce task handles all (k, v) pairs with the same key
- Keys are passed to the task *in sorted order* (but things may be scattered among multiple reducers depending on how you partition)
- Why? 1: Friendly to humans, predictable/deterministic output; 2: many apps need sorted output for later stages (searching); 3: Have to do grouping by key *anyway*, and sorting is a convenient way to do that. Especially given good external sort algorithms.
- Cool trick: distributed sort using MapReduce? map(k, v) -> (k, v)
  reduce(k, list<v>) -> list<v>
  partition() by range

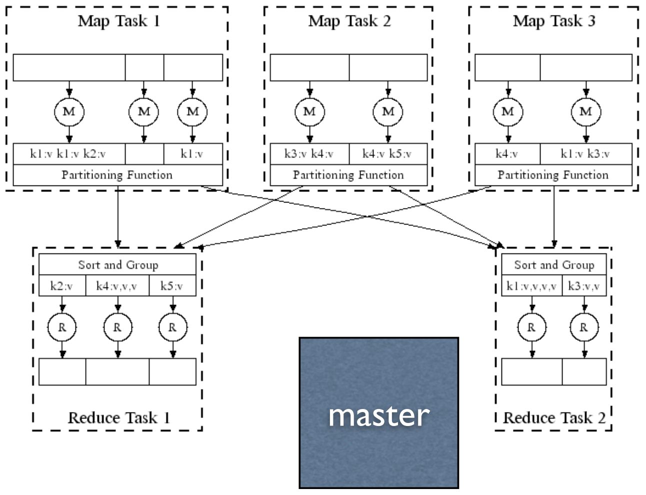

figure from Jeff Dean OSDI talk slides

- Okay, that makes sense.
- But how do you *do* it?
  - Where is data stored? Where do you execute map and reduce tasks? How many? Failures? Load balancing? How to get (k, v) from one stage to the next?

Data Storage

- Data stored in ... a cluster distributed filesystem (GFS, HDFS)
- Each node has both storage and computation
- Data is replicated ~3 times to deal with failures, help performance
- We'll talk about the filesystem next time. For now:
  - Data stored on machine X can be accessed faster by machine X than by other machines (locality helps), but
  - Any machine can access any data item if it wants
- Input comes from this FS, final output goes to it

Work flow

> When a user program starts, it starts up MapReduce. One of the most important early steps is for MapReduce to carve up the input file(s) into chunks, known as *splits*. Each split is of the same size, which is user configurable anywhere from a dozen to several dozen megabytes.

MapReduce then initializes a whole bunch of instances across many nodes. One of these instances is the *Master* that is responsible for coordination. The other instances are *Workers* that will each perform Map and/or Reduce operations. The Master will assign idle workers Map or Reduce tasks.

But, it does not assign more than one task per worker. If there is more work to be done than workers available, the Master will hold onto it until some Worker becomes idle and can immediately accept it. By keeping the de facto work queue at the Master, rather than on the Workers, the Master is able to improve load balancing. This is because Workers will likely finish at unpredictible and different times, making it hard to optimally allocate all work initially.

The Map Worker does its thing and churns out the results -- the intermediate key-value pairs. These results are buffered in memory for a while, but periodically written to disk. As they are written to disk, the key-value pairs are hashed into *Regions*, based on their key. The data is divided into Regions to provide chunks that can be processed in parallel by Reduce workers. By dividing the output using a hash function, the buckets associated with each worker will be approximately the same size.

As each Region is written, the Master is informed. This allows the Master to assign the work to a Reduce Worker, which will read the data from the intermediate file using an remote read, such as by RPC call.

Allocation to map tasks and reduce tasks

- Assume that more nodes is better (for big tasks).
- Depends on ratio of computation time between maps & reduces (and the "hidden" sorting stage before invoking reduce)
- More map tasks means more input parallelism (good), but want input chunk size to be large enough for efficient reading (16--64MB is common)
- For load balance, want more reduce tasks than processors (a few times more)
- But more reduce tasks means more output files, which you may have to aggregate later
- goog example: 200,000 maps, 5000 reduces, on 2000 machines

Locating maps & reduces

- Locality helps when reading from the dist. FS
- So assign map tasks to data stored locally on that node, *when reasonable*. Alternately, you might allocate to nodes on same rack as reducers if there's more bandwidth between nodes in the same rack than across racks
- Reducers probably take input from lots of maps, so locality less important.

Optimization: Intermediate aggregation

- Often reasonable to do an intermediate "reduce" phase before shipping across network. For word count example, can actually reuse the reduce function - aggregate ("hi", 1), ("hi", 1) into ("hi", 2) before sending across network to reducer
- These "combiners" must be commutative and associative (another restriction on the problem that lets you do more powerful things)
- Lots of fun researchy questions about how to best do this. Bumps into database literature, too.

Worker Failure

- Master periodically pings workers.
- If you can't talk to a mapper for a while, just re-execute its jobs somewhere else.
- Handling reducer failure: Have its final output commit be atomic (it's either all there or all gone).
  - Usual trick: output to temp file, close it, and then atomically rename it
- Slow nodes: Just have someone else re-do the work, let them race to finish. Good for load balancing. ("end-game")

Master failure

- Google's first answer: meh, don't worry about it, just re-execute the entire job.
- Google's new answer: replicate the master using Paxos. :)
- Yahoo!/Hadoop's first answer: don't worry about it.
- Hadoop's new answer: ... paxos ...

Doing Complex Things

- Many computations can't be expressed as a single Map +Reduce phase
    - End up building multiple stage pipelines
- Google engineer's perspective: Often, you start out thinking "i can do this better than using mapreduce", but almost always fall back to using MR because it handles so much ugly stuff for you that you'd otherwise have to reinvent

Going further

- GOOG & YHOO both have higher-level languages to make simple data analysis using MR easier-- Google's "Sawzall" and Yahoo's "Pig"
- MSFT generalized to "Dryad", which lets you set up pipelines more flexibly instead of just doing maps & reduces (but maintains same commutativity, side effect freedom, etc)