MapReduce implementation for Python and NetworkSpaces

Term Project for Parallel Programming Techniques

Antonis Stampoulis
antonios.stampoulis@yale.edu

Outline of this presentation

- The MapReduce framework
- Implementation of normal MapReduce
- Sample programs and performance results
- Implementation of adaptive MapReduce
- Discussion of overhead for adaptive version

What is MapReduce?

- general framework for distributed programs
- two operations: map and reduce
- map: run on instance of input, produce intermediate key-value pairs
- reduce: run on group of key-value pairs with same key, produce final key-value pairs
- in general, the main operation is the map; the reduce is usually simple

Sample program: dictionary grep

- Match a regexp against a set of files, return matching substrings + count of matches

```
map(emit, fn, regexp):
        f = file(fn)
        contents = f.read()
        matches = regexp.findall(contents)
        for i in matches:
            emit(i, 1)

reduc(emit, k, v):
        emit(k, sum(v))
```

Using MapReduce

What about input?

- Of course we need to specify the input-set where the map operation will be executed
- Partitioning of the input will be essential!
- No easy way to generalize partitioning – having the user provide the full input set plus a metric for each instance isn't always a good idea
- Solution: have the user of the MapReduce runtime specify the partitions themselves!
- The user can do this in the original MapReduce implementation as well
- We need another function, called 'gen', which *yields* all the partitions.

Calling the MapReduce runtime

- No surprises! Just call the relevant function.

```
r = mapreduce.mapreduce(
        workers = ['frog', 'gator', 'hippo'],
        module = dgrep,  # module containing our code
        gen_params = [directory, 32*1024*1024],
        map_params = [re.compile(pattern)],
        reduce_params = [],
        reduce_assoc = True, # is reduc associative?
        reduce_tasks = 6)       # if not, how many tasks?
```

Implementation of normal MapReduce

- Normal Python / NetWorkSpaces / Sleigh program
- Master / worker, with one or two worker phases (we'll see later why)
- Map phase:
  - agenda parallelism (workers grab next available task and execute it)
  - each task is one partition, as provided by the gen function
  - execution of each task is easy – just loop through the partition, and execute the map operation for each instance of the input in the partition

Implementation of normal MapReduce

The Map phase in more detail

- How are tasks generated and handed to the workers?
  - One possible way: have the master use the gen function to generate all the partitions, and store each partition into the network space as a task, for workers to grab
  - Problems: non-parallelizable overhead of generating the partitions in master, large task descriptions
  - Solution: use something like the DIY trick!
  - Master stores just the first ticket
  - Workers grab ticket, increase it, and use the generator object to get the partition described by the ticket
  - When no more partitions: worker poisons workers!

Implementation of normal MapReduce

The Map phase in more detail

- How does emitting intermediate key-value pairs work?
  - Naïve solution: emit(k, v) = nws.store('result', (k,v))
  - Problems: tremendous coordination overhead! (many small-valued stores >> a few large-valued stores)
  - Also: we can't do any extra local processing of the intermediate pairs
  - Solution: the emit instruction caches the key-value pairs locally in a dictionary, associating each key with the list of values corresponding to it

Implementation of normal MapReduce

The Map phase in more detail

- What happens next?
  - **If the reduce is associative**, we can locally perform the reduce operation on our intermediate results!
  - Cache the results of the reduce operation and post them all together back to the network space
  - Usually reduce **is** associative, and also the resulting pairs from reduce are of smaller size (because usually reduce summarizes a list of values into one value)
  - Double gain: less communication, less work that remains to be done (reduce intermediate reduce results)
  - this work is little enough that executing it on the master has less computational content than the coordination overhead of parallelizing it!
  - we execute it on the master and MapReduce is done!

Implementation of normal MapReduce

The Map phase in more detail

- What if reduce isn't associative?
  - the work that remains to be done might be a lot, so we need to parallelize it
  - We want to split the intermediate pairs into R chunks, where R is the number of reduce tasks we want to have
  - Easy: group intermediate (k,v) pairs into R buckets, using a hash function that returns 0 to R-1
  - Important: the hash function must give the same result for the same input on all nodes! (we can't use built-in hash)
  - Finally: each worker node posts its part of the R buckets into the network space

Implementation of normal MapReduce

The Reduce phase

- the master posts descriptions of the reduce tasks (task description = number of chunk to be processed), waits for them to be done, and gathers results
- the workers:
  - grab a task
  - fetch all the values of that chunk of the intermediate key-value dictionary from the network space
  - group key-value pairs by key (using dictionary)
  - perform reduce operation on each key-list of value pairs
  - the emit operation caches results into result dictionary
  - when all are done: post result dictionary to the network space

Sample programs

- We have three sample programs for MapReduce
  - *dictionary grep*: match a regexp against all files in a directory (recursively), return the set of all matching substrings, together with the number of times each substring matched
  - *maximum line*: find the maximum (lexicographically) line out of all the files in a directory
  - *prime number finder*: find all prime numbers up to N (just as in programming assignments for the course)

Performance results
Dictionary grep

- When input files are cached:
  - Serial: 32.5 sec
  - NWS (3 workers): 14.3 sec (75% efficiency)
  - MapReduce (3 workers): 14.4 sec (75% efficiency)
  - MapReduce (5 workers): 10.1 sec (64% efficiency)
- Analysis:
  - Ts: startup (0.3s), generate partitions (1.51s), reduce operations on master (0.12s)
  - Tcs: open sleigh (0.57+W*0.03s), fetch map results in master (0.093s)
  - Tp: do map operations (30.49s), local reduce (0.12s)
  - Tcp: store map results (0.25s)
- For serial case: very accurate! (Tcp = Tcs = 0)

Performance results
Dictionary grep

- Is this model accurate for the parallel case?
  - This model was devised from the W=5 case, by getting accurate timings of all the phases
  - Total Runtime(W) = Ts + Tcs(W) + (Tp + Tcp) / W
  - Predicts run-time:
    - W=3 case: 12.97sec (compare to 14.4sec)
    - W=5 case: 8.92sec (compare to 10.1sec)
  - Why these are off?
    - (Tp + Tcp) / W is not accurate – it would be accurate if all workers get equal part of the input set
    - in reality, one worker does more work. If we take Tp + Tcp = W * (max{Tp + Tcp} for a specific run), we get more accurate results (14.97s and 10.11s respectively)

Performance results

Prime numbers

- Primes up to 500000, Chunk size 5000
  - Serial case: 13.6s
  - MapReduce (3 workers): 8.6s (eff. 52%)
  - MapReduce (6 workers): 5.7s (eff. 40%)
  - NWS (3 workers): 7.6s (eff. 60%)
  - NWS (6 workers): 4.8s (eff. 47%)
- Analysis
  - Ts = startup (0.2s), generate input (0.23s), reduce operations in master (0.18s)
  - Tcs = open sleigh (0.87s), fetch map results in master (0.14s)
  - Tp = do map operations (20.58s, max: 7.02s), local reduce operations (0.12s)
  - Tcp = store worker results (0.3s)
- Predicted time: W = 3: 8.7s, W = 6: 5.3s

Performance results
Prime numbers

Ts = startup (0.2s), generate input (0.23s), reduce operations in master (0.18s)
Tcs = open sleigh (0.57s), fetch map results in master (0.14s)
Tp = do map operations (20.5s, max: 7.02s), local reduce operations (0.12s)
Tcp = store worker results (0.3s)

- Why is NWS faster by ~1sec?
  - small variations in inner loops have significant effects in running time: by inlining the map operation, Tp is reduced by 0.5s
  - also, reduce is redundant, startup is less (no MapReduce compilation), and coordination overhead smaller (lists are smaller than dictionaries!)
- Why is the core of the computation slower than the serial case?
  - again, because of (unavoidable) variations in inner loops

And now for something completely different...

Sleigh

(non-adaptive: specify all workers in the beginning, everybody executes tasks)

BobSleigh

(adaptive: potential workers determined in the beginning, then they dynamically decide whether they want to execute a task or not)

Implementation of Adaptive MapReduce

The BobSleigh architecture

- BobSleigh server
  - normal NetworkSpace in a decided-upon host
  - keeps info about potential workers, bobsleigh tasks, whether a worker is allowed to work or not, etc.
- BobSleigh client
  - daemon that runs on every potential worker
  - register potential worker with BobSleigh server
  - waits for new bobsleigh task to appear
  - then launches the control process for that task: automatically (and dynamically) chooses whether worker should take part in computation or not based on load level and X event idleness
  - user can override the decision

Implementation of Adaptive MapReduce

The BobSleigh architecture

- How do we store whether a worker is allowed to work or not?
  - variable per worker and per task, that is only assigned a value when the worker is permitted to work
  - also another variable that controls whether a worker node is still allowed to work while computing a task
- How is this architecture used?
  - use BobSleigh class instead of Sleigh, which initializes a Sleigh with **all** the potential workers
  - all the potential workers execute the initialization/de-initialization code
  - we use the allowed variable to limit execution of the core computation

Implementation of Adaptive MapReduce

Changes to the implementation

- Map phase
  - pretty straightforward: instead of waiting for a ticket, first wait to be allowed to work! (decided by BobSleigh client)
  - then grab a task and execute it
  - check periodically (in the inner loop) whether we're still allowed to work
  - if we're not, bail out of the computation, but augment the task description with the place where we left it (easy)

- What does this change in the master?
  - we need to know how many tasks there are beforehand
  - thus master needs to generate partitions, so he posts the tasks too (no DIY-like trick)
  - when all tasks are done, poison **all** workers

Implementation of Adaptive MapReduce

Changes to the implementation

- Reduce phase
  - in the associative reduce case, nothing changes!
  - when reduce is not associative, we have a second round of BobSleigh tasks, where we use the per-worker allowed variable to control whether a worker grabs reduce tasks or not
  - but: we don't permit workers to bail out of a reduce task once they grabbed it; stopping a computation would require posting back the intermediate results (the coordination overhead of stopping would be greater than the computational content of completing the task)
- What about the user code?
  - No changes needed at all – adaptive parallelism is free!

Implementation of Adaptive MapReduce

The BobSleigh client, revisited

- How does the BobSleigh client make the decision when to take part in a computation?
  - checks every 1 sec for load **of the other processes**, and for the time since the last X input event
  - when the load of other processes is < fixed value, and when the idle time is > fixed value, start computation
  - when either of these doesn’t hold, stop the computation
- We can do this because the overhead of starting and stopping the computation is very small!
  - essentially: if we have a task, abandon it, and someone grabs it, the overhead is minimal, because they continue just where we left off

Discussion of overhead for the Adaptive version

- What extra overhead does this implementation impose?
  - Master:
    - has to generate all input partitions, just to get their count! (this is done twice, once in master and once in all workers – if master were to post these partitions to the workers, the coordination overhead is larger than the partition generation cost)
  - Workers:
    - overhead of periodically contacting the BobSleigh server to check whether they should stop computing
  - BobSleigh client (again in workers):
    - Overhead to create the control window (to permit users to make their decisions)
    - Overhead to monitor the load and idle times

Discussion of overhead for the Adaptive version

- Results:
  - We ran the same tests in the same machines, setting the BobSleigh client to always take part in the computation
  - Prime number finder is ~ 0.6 sec slower (input generation takes 0.23sec, the rest is window creation)
  - Dictionary grep is ~1.8 sec slower (input generation takes 1.51sec, time for window creation is almost the same)
- What about degradation of user experience?
  - pretty small: within 1sec of input event or significant rise of load of other processes, we stop using any cycles
  - after all map tasks done, we still need to do the reduce on local intermediate results, and post the reduce results to the network space – but this is in the order of a few 100msec’s

Thank you very much!

Backup slides

What is MapReduce?

- pros
  - simple principles, elegant code
  - coordination is mostly abstracted away
  - can be made efficient in various settings
  - optimizing/adding features once (in the runtime) instead of in each program
- cons
  - not every application is a good fit
  - assumptions made in the runtime may not be true for all applications (e.g. reduce might be costlier than map!)

Sample program: dictionary grep

- Split all files in a directory in partitions more-or-less equal to the given chunk size.

```
def gen(path, chunk_size):
    current_size = 0
    current_partition = []
    for root, dirs, files in walk(path):
        for fn in files:
            name = join(root, fn)
            size = getsize(name)
            current_partition.append(name)
            current_size += size
            if current_size > chunk_size:
                yield current_partition
                current_size = 0
                current_partition = []
    if current_size > 0:
        yield current_partition
```

Using MapReduce

- First of all, we need to specify the map and reduce operations
- Functions that are named ‘map’ and ‘reduc’ in a module
- Arguments for map: the emit function, the current instance of the input, plus user-specified arguments
- Arguments for reduc: the emit function, the current key – list of values pair, plus user-specified arguments

Implementation of normal MapReduce
The Map phase in more detail

```
while not done:
    ticket = SleighNws.fetch('map_phase_ticket')
    if ticket == None:
        SleighNws.store('map_phase_ticket', None)
        break

    SleighNws.store('map_phase_ticket', ticket+1)
    while chunk_num < ticket:
        try:
            chunk = chunk_gen.next()
            chunk_num = chunk_num + 1
        except StopIteration:
            done = True
            # poison the other workers
            SleighNws.fetch('map_phase_ticket')
            SleighNws.store('map_phase_ticket', None)
            break
    # ... process this partition
```

Implementation of normal MapReduce

The Reduce phase

- If the reduce operation is associative
  - we have the results of the reduce operation on the intermediate key-value pairs in the network space
  - one dictionary of k => v pairs for each worker
  - What needs to be done?
    - fetch all dictionaries
    - for each k, perform reduce on all associated v’s (at most as many as the workers)
  - We don’t parallelize this – master does all of it (coordination overhead > computational content)
    - overhead: workers have to split their intermediate dictionaries using hashing, master has to hand out tasks, workers have to store result + master has to read it

Sample Programs
Dictionary Grep

- We saw this already:
  - gen: partitions of almost equal bytesize of the filenames in the directory
  - map: for each matching substring in file, emit key = substring, value = 1 (count one occurrence of that substring)
  - reduce: for each key = substring, list of values = list of number of its occurences, emit (substring, sum(occurrence_list))
  - reduce is associative!

Sample Programs

Max line

- Similar to dictionary grep, less communication:
  - gen: partitions of almost equal bytesize of the filenames in the directory
  - map: read file, emit key = ‘max’, value = max line out of file
  - reduce: only one key: ‘max’, list of values = max lines of a set of files. Emit ‘max’, max(max_lines).
  - reduce is associative!

Sample Programs
Prime number finder

- Easy:
  - gen: chunks of numbers of equal size
  - map: for each number, test for primality (with utterly naïve test), and emit (number, True) if it is prime
  - reduce: redundant; just emit the (key, value) pair back – this is because there's only one value for each key
  - reduce is associative again!