Athena

Purpose

This document gives an outline of what project I am proposing for an independent study in the spring of 2013 with Professor Samik Basu. In summary, I would like to build a **resource manager/job scheduler/distributed threadpool for highly parallel time-slice sequential analysis jobs that are stored on a Cassandra cluster**. This proposed tool would be named Athena, and it is presumed that Athena a would run on a modern Linux platform (2.6.x Debian with v3.4 Kernel for this project).

An Example of Parallel Time-Slice Sequential Analysis Workloads

Consider a security company that stores data from 1,000,000 close circuit cameras installed at large airports throughout the world. Records are never erased and so the amount of data expands quickly, leading them to adopt a NoSQL datastore solution. Automatically generated programs are constantly searching for recognizable patterns in the time slices surrounding known criminal events, and comparing those patterns to other time slices\* of footage. Each time slice can be considered a collection of images, but these images must be analyzed sequentially for the algorithm to make sense of them. The need to analyze the data sequentially makes it difficult or impossible to apply the MapReduce model to these analysis jobs, (which requires that the result of each key-value input to the map function be independent of the other results). Each job can be done independently of all the other jobs, allowing for significant speedup on a cluster with N independent processing cores.

(\* an inquisitive reader may wonder what specific time slices are being selected. Assume for this discussion that these time slices are evenly distributed throughout the possible range, but in practice there may be hotspots of data. One possible mitigation of the hotspot problem is to increase the replication of hotspot data, but for simplicity, this project will leave that out.)

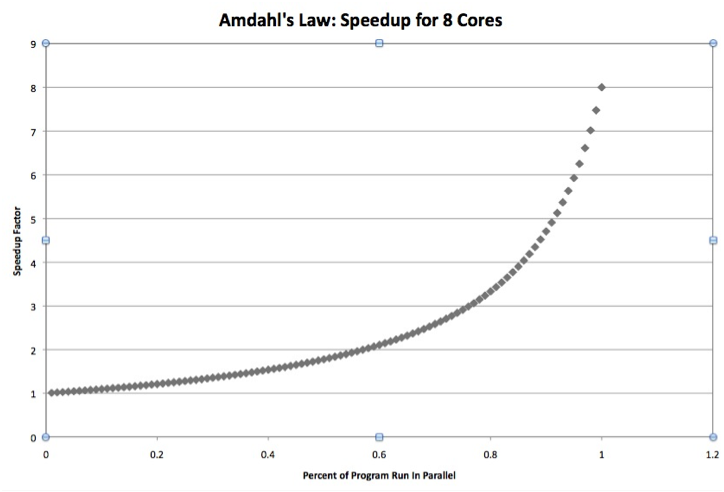
Speedup In Parallel Analysis

When considering a program that could be written to run on one core or N cores, the theoretical maximum speedup is N times faster than one core. The actual speedup (S) of a multithreaded program is given by Amdahl’s Law [13] :

S = 1 / ((1 - P) + P/N)

Where P is the percent of a program that can be run in parallel.

The following table summarizes possible speedups for different values of P with N held fixed at 8. Each point is a 1% increase in P. Note that the graph would be the same shape for any number of cores.



The point of looking at this calculation of speedup is that the most significant speedup happens in the last few percent increases in parallelism. Athena is intended for applications that do extremely parallel calculations, trying to take advantage of Amdahl's law. Ideally, the work of distributing tasks among nodes will be done in parallel with the analysis work being done. Blocking the tasks to coordinate communication decreases parallelism and so will be avoided.

Goals For A Distributed ThreadPool

On a single machine with multiple cores, threads are used to execute runnable tasks in parallel. If there are a lot more threads than cores, the overhead of switching between threads can consume significant resources, depending on how many threads are waiting on I/O, etc. A threadpool is a collection of threads that can be sized dynamically to minimize thread-switching costs, while handling the responsibility of executing a queue of tasks of arbitrary size.

Athena will create a threadpool on each node and manage the generation and distribution of tasks - and is therefore a distributed threadpool. (Notice that generation of tasks will be done automatically and on all nodes - it is assumed that these tasks will require some arbitrary data types, and cannot be constrained to require data on their parent node.) The primary goal of Athena is to try and **maximize the rate of productive computation**. This involves several goals:

1) minimize internode communication costs

*  avoid passing tasks when their parent node has required data
* send jobs to data (localization) - this minimizes communication because it is assumed that in general, results will be smaller than the data used to create results
*  optimize message passing - Athena will be using a ‘pulse’ communication strategy for this purpose.
*  minimize management overhead - make the communications system as efficient as possible

2) manage size of data in memory (notably task input data and results data)

*  use fast SSD disks to store large datasets out of memory on volume groups:

1. input data passed to a task from another node
2. results in progress and to be sent at the next pulse
3. received temporary results
4. final results (created by combining received temporary results)

* prevent stackoverflow - (see table below)

3) maximize task time on all CPUs

*  at each communication pulse, plan work generation and distribution (see table below)

The following table describes all of the optimizing strategies as functions with input and output. All functions except the first are calculated during the part of the pulse cycle called ‘consensus’, which is green in the pulse cycle diagram (so they are green here as well). In brackets there are qualifiers stating when parts of functions are calculated; [dynamic] means it is calculated all the time as work is being done, [static] means it is a user-configured value, [consensus] means it is a calculation done at consensus time. Since these functions depend on each other, they are done in a particular order (top to bottom in the table). There are also all-caps phrases that give the general idea of what’s going on for that function.

|  |  |  |
| --- | --- | --- |
| Concern | Function input | Function output |
| Node CPU task utilization vs thread scheduling costs | Percent of cpu time used by tasks (sched\_debug) AND min thread number | Node Threadpool (TP) size [dynamic] |
| All objects become too large for JVM on a given node | recent peak memory size of heap before garbage collection (per Node) [dynamic] AND ceiling size [static] AND recent minimum number of total tasks held in queues and the threadpool [dynamic] (mixed with recent max number of task figures) | Map of available Space Capacity as maximum number of tasks CAPACITY PLANNING [consensus] |
| Tasks overflow SSD partitions | results received and final results partitions size map [dynamic] | Partition Map (no computation) |
| Node CPU idle time due to having no tasks | Recent rate of work performed [dynamic] AND Current Queue work Size [consensus] AND last waitTime [consensus] (plus small buffer?) | Map of Orders ORDER SIZE |
| System CPU idle time due to cold spots and regular consumption | Map of Orders [consensus] AND data Localities [consensus] AND available Space Capacity [consensus] AND map of network latencies [dynamic] AND Task generating object Data Locality requests [consensus] AND Partition Map | Map of work shipments PLANNING |
| Actually shipping work to nodes | Map of work shipments AND collection of Task generating objects | Outgoing/local starting queues filled up. Messages passed SHIP |
| Minimize communication time / coordination | Map of all filled queues AND historic work performance rate | New startTime |
| Uneven distribution of communication needs | longest time in last cycle where one process communicated results while any other process was finished communicating results. (mixed with previous naptimes) | new napTime |

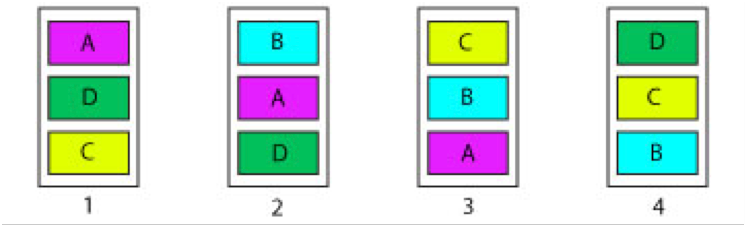
Some of these functions involve mixing previous results (PR) with the current result (CR) in an attempt to stabilize the number over time. Here is a proposed algorithm to get a new result (NR):

NR = ((CR + PR + PR + PR)>>)>>

This is just an attempt at performance optimization using bit shifting. Perhaps it will backfire. Unit tests will tell.

Data Localizing With Cassandra

Cassandra is a distributed NoSQL datastore solution that runs in Java on a network of machines (cluster). Each node in the cluster (1,2,3,4 in the diagram) is responsible for a set of Cassandra tables (A,B,C,D in the diagram), which are replicated throughout the cluster to a certain degree (for our example, R=3). Let's say that each table stores a certain type of data, and so for each node, certain data types are local.



For example, on node 1 in the above diagram, data types A, D and C are local. When a query is made in a process running on node 1 for data of type B, Cassandra's proxy service will find a source for data of type B. The data of type B must be communicated over the network, which could use a substantial amount of resources. If the task that needs type B data could be sent to a machine with type B data, the network transfer costs could be minimized. For this project, an 'OrderedPartitioning' scheme is used, meaning that each table partition stores a sequential range of data, and as a result, returning a time slice is efficient. Most Cassandra deployments use a random partitioner to help balance loads and it’s often recommended to store parts of time series in columns. This would defeat Athena’s attempts at doing manual data localization.

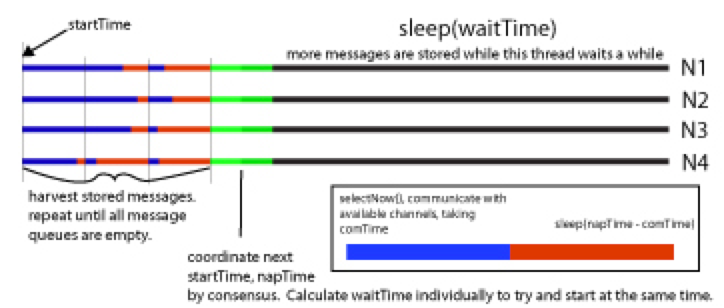
Non-Blocking

Communication in a distributed network can be blocking or non-blocking. A blocking approach might make a call to open a socket, blocking the thread that opened the socket until the response is received (via interrupt) from the socket on the other end. Because the waiting thread is blocked, the CPU scheduler must then swap another thread onto that core. When the sending socket opens it sends a message saying that it has opened. The listening socket thread is in a waiting/blocked state and the CPU scheduler must finish its epoch and then at the start of the next epoch it will check wait conditions for all blocked threads and if it has had a signal from the other end of the socket, it will wake up the waiting process and allow communication [2]. This means that in a blocking network, all communications should probably happen on separate threads so that sockets ready to go are not blocked by one that is waiting or perhaps broken. Of course while I/O is happening, the thread is blocked until communication ends.

With a 'full mesh' type cluster of N nodes where every node connects with every other node, each node will need N-1 separate socket threads. For tiny clusters, this is fine, but for a large cluster, perhaps of 40 nodes, there would be the overhead of 39 blocking threads on each machine (1560 threads total). Running this many threads all waiting on other remote threads could waste a non-trivial amount of resources just to do scheduling/swapping in and out. So one of my theories is that blocking communication does not scale well [1].

Selectors and the Pulse Communication Strategy

The Selector class provides a way to poll the state of sockets without blocking, so a single thread can get many connection-related tasks going without being blocked. Most notably the 'selectNow()' method can immediately return all sockets ready for communication (sockets on the other end must be SelectableChannels set to nonblocking). When the Selector finds sockets that are ready for communication, then it can send or receive data. My approach to reducing communication overhead is to pulse communication based on some kind of dynamic timing scheme. Since bandwidth is fairly ample, messages can be queued up for a period of time and then sent all at once at an agreed-upon time. The fundamental reason that I think a pulse strategy will improve performance is that it follows the concept of economy of scale, like the difference between delivering coal by train vs. by UPS truck. Most distributed network frameworks that I've found focus on being responsive and event-driven, perhaps because they focus on web applications. For a private network/cluster focused only on efficient use of resources, I think that a pulsing framework is a better fit.



Sequence of Events

1. Before Startup

* All nodes have a configuration file with default values for:

1. threadpool size
2. waitTime (~1sec)
3. napTime (1ms?)
4. map of node IPs to:
5. type of data and time range for that node
6. Athena JVM heap size (and protective ceiling size for each)
7. SSD lvms size for the four partitions (data, results in progress, results received, final results)

* The following services are already instantiated:

1. Cassandra containing data
2. Athena communication services and threadpools, sockets set up with nodes listening to each other, pulsing heartbeat signals, say, every second.
3. Factories that create and manage task-generation objects (reachable through communication services)
4. One node with a central manager, ready to send out a start signal.
5. Initializing

* Using cmd line on the node with the central manager, the (initial) start signal is sent out to all the nodes to start generating tasks.
* Factories on each node create a number of task-generating objects capable of filling up the final results partition on that node [after all task generating objects complete tasks and combine intermediate results into final results (the total size of the intermediate results may be many orders of magnitude larger than the final results)].

1. Base Cycle

* pass back returned results if any (bulk of communication), erase intermediate results partition.
* If a task generating object gets all the results it needs to create a final result, it does so and erases all of the intermediate results on the partition that it no longer needs
* achieve consensus [in real time] - gather information about the cluster and decide what tasks to do locally, what tasks to ship to data, what tasks to do that require a mixture of the above and what tasks (if any) to ship to a node that will not contain the desired data. Reaching consensus involves the functions listed above. (For the first round, tasks will be assigned until each node has at least one task. After that round provides input for consensus functions, a conservative estimate will be made as to how many tasks to assign to keep the node busy but not overfilled. As the system ‘warms up’, it should get better at achieving an optimized distribution of work.)[maybe do ‘trim’ at the same time as concensus]
* Ship whatever tasks need to be shipped, if any. Sleep the communication services until the next startTime.
* Nodes go about the business of doing tasks on their pool. Each task keeps under a maximum size and writes intermediate results to disk as needed.

1. Meta Cycle

* Once all task generating objects have written to the final results partition, it should be nearly full. These final results are written to Cassandra, the partition is erased and the entire process may be repeated again with completely new and different task generating objects. It is assumed this will involve some genetic programming but is not under the purview of Athena.

Potential Problems and Robustness

* Tasks might generate exceptions when they are on the threadpools, and that will be handled by sending a result back to the task-generating object indicating the exception.
* some tasks will be lost in transport - UUID tracking, missing task restart (n times)
* get into an infinite loop - in-threadpool updates and timeouts
* garbage collection will cause bad performance - tuning GC and scheduler
* memory will overflow - space calculations and writing results to disks.

Other Thoughts on Implementation

Decent random numbers can be generated quickly using the XORShift algorithm, bitmasks, etc.

Also…try different linux scheduling algorithms in permutation with relevant JVM GC options.

Also...to avoid results data getting too large. Some ideas about this: each fetchable row of data will aim to be about 1MB in size. (Tasks that need more than one type of data and/or cannot colocate can put fetch and temporarily store their data in a special SSD partition.) Tasks will stream through this information at about one row at a time, appending results (n >= 1MB at a time?) to a file on one of a pair of SSD partitions (10+GB each?) used exclusively for these temporary results files. This file I/O is extra work, but it will allow tasks to create large intermediate results data without danger of exceeding the JVM heap size. Care will be taken when planning to make sure that these partitions contain enough room for the expected results sizes, but if both temporary results partitions fill up, the threadpool can react somehow - perhaps all tasks pause until the next communication pulse. When a task is finished it writes the return message into the returning results message queue which keeps a reference to the file it wrote. When passing the message back, the communicating I/O streams the file from the sender to a receiving file in a similar redundant SSD (again, 2 x 10+GB on receiving end) partition to protect the receiving memory size in the same way. Then once it is finished the sending machine deletes the temporary file. Similarly, the task-generating object eventually consumes and deletes the results file, storing final results in another pair of SSD partitions. These SSD partitions must be periodically trimmed to keep them from slowing down.

Testing Plan

I'd like to have several distinct large datasets that do not fit in memory being stored on each test node. For each dataset I'd like a calculation task of randomly varying difficulty that can be serialized as a message and passed to Athena's thread pool interface. Each test session will have a certain number of task-spawning objects which each spawn a specific number of tasks. The intention of this test strategy is to simulate a never-ending analysis workload, and see how the various self-tuning features of Athena work on a large scale.

If practical to implement, there should also be a few control programs competing with Athena, running identical tests. They would replace Athena's non-blocking communication module (a Netty implementation might be competitive), or self-balancing threadpool module or data localizing module (or combinations of these) with an alternative implementation that makes no effort to optimize. This might help to identify parts of Athena that are most and least beneficial.

For each of these competitive tests, it makes sense to test different communication workload scenarios. For example, one simple scenario is where all datasets are equally in demand, and all tasks are about the same size. The range and distribution of task sizes and the overall workload corresponding to each dataset should be varied to see how optimizations are affected by these factors.

The time it takes for all task-spawning objects to complete all of their tasks is measured for each test session. Total communication time, CPU utilization, network traffic load and number of lost tasks may also be used to measure performance.

References

[1] 'thundering herd problem' http://www.linuxjournal.com/article/8144?page=0,2 AND http://www.cs.columbia.edu/~smb/classes/s06-4118/l13.pdf AND http://natishalom.typepad.com/nati\_shaloms\_blog/2008/10/is-mapreduce-going-to-main-stream.html

[2] http://oreilly.com/catalog/linuxkernel/chapter/ch10.html ('The Scheduling Algorithm') AND http://www.cs.unh.edu/cnrg/people/gherrin/linux-net.html (Chapter 6, 'Receiving messages') This second resource explains the top-half, bottom half concept that relates interrupts with the scheduler. Simply put, an interrupt does not immediately stop everything right away, but raises a flag that there is something to do and then returns control. When schedule() is run again for that process, it checks flags and will know to do something.

[A] an interesting look at java garbage collection problems and performance prediction: http://www.cs.rice.edu/~zoran/Publications\_files/latex8.pdf AND http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html AND https://blogs.oracle.com/jonthecollector/entry/our\_collectors