TaskSprint: A general purpose distributed computation system

Sergio Benitez Ivan Sergeev Alec Thomson

1 Introduction

TaskSprint is a general-purpose, fault-tolerant, distributed system for deterministic and non-deterministic computation. TaskSprint differentiates itself from existing systems, like MapReduce and Pig, by allowing non-deterministic computations, allowing arbitrary stage computations (as opposed to two in MapReduce), having no central point of failure (as opposed to Pig/MapReduce's master), being language agnostic, and being computation agnostic.

TaskSprint's core challenge is to provide all of these facilities in a fault-tolerant manner while exposing minimal complexity to the user. To this end, TaskSprint's only requirement from the user are two small pieces of software: a scheduler and a node. The scheduler's job is to define a schedule for tasks, and the node's job is implement those tasks. All the intermediate work is handled by the system. This includes distributing and replicating the computation across nodes in the network, handling data shuffling efficiently, handling task dependencies, and dealing with scheduler, node, and network failures.

2 Design

TaskSprint's infrastructure is centered on two major components: the coordinator and the client. The coordinator communicates with the user's scheduler, and the client communicates with the user's nodes. The coordinator and the client communicate with each other to execute the user's computation. To support replicated, non-deterministic computations, we seed PRNGs with the same value.

2.1 Coordinator

The coordinator is a Paxos-replicated, event-driven state-machine that responds to clients' requests for the current task schedule. Clients query the coordinator for the latest "view", the task schedule, and notify the coordinator when tasks are completed by nodes. A view includes a mapping of tasks to clients in addition to task parameters as defined by the user's scheduler. Parameters include task name, task arguments, and task dependencies.

The coordinator monitors the liveliness of a clients. When a client hasn't queried the coordinator for a certain period of time, the coordinator marks the client as dead and reassigns its tasks. A single replica known as the "leader" periodically inserts "TICK" operations into the Paxos log to ensure that all replicas exactly agree on when clients died in relation to other operations. New leaders are periodically elected via Paxos to handle leader failure.

The coordinator communicates events to the user's scheduler. In response to events, the scheduler can start new tasks, kill existing tasks, or end the computation. These event handlers are very simple and easy to implement while providing developers a large amount of flexibility for their computations. For instance, a genetic algorithm scheduler is able to group similar solutions for crossover and a Bitcoin mining scheduler is able to kill tasks to free resources when solutions are found.

2.2 Client

The client's main tasks are to launch nodes, schedule tasks on nodes, and handle task dependencies, which may involve efficiently transferring data between nodes and clients. To launch nodes, the client detects the amount of available resources on the machine it is running on and launches a number of nodes proportional to the amount of available resources. The client-node communication is done via IPC using a language-agonistic, JSON driven protocol.

A client is alerted of new tasks by polling the coordinator for the most recent view. The client checks for changes to task assignments, finds tasks intended for it, and identifies any dependencies for its tasks. If there are dependencies, a client must satisfy them before executing the task. To do this, the dependent client communicates with the supplying clients responsible for the pre-requisite tasks. If nodes on a supplying client have completed the pre-requisite task, the needed data, if any, is transferred to the dependent client. This is done on a rolling basis such that data is transferred asynchronously as soon as a pre-requisite tasks are complete.

Once the client has obtained the necessary input data for a task, either by satisfying dependencies or directly from the scheduler, the data is serialized according to the JSON driven, language-agnostic protocol and the task is executed on an available node. Upon completion, a node notifies its client with the results.

The client stores the results so that is may satisfy future dependencies and notifies the coordinator about the task's completion.

3 Examples

3.1 Multiple Root Finder

A root-finding algorithm finds x for which a function $f(x)$ is 0. While many root-finding algorithms exist, few are equipped to search for multiple roots of a function without new instantiations of the algorithm initialized with different initial guesses. Our distributed multiple root finder searches a wide solution space stochastically, obviating the need for initial guesses, and evolves multiple clusters of the "fittest" roots to likely roots within a specified epsilon. The search is defined by a simple set of parameters, including: a function of an arbitrary number of variables, a bit resolution, solution space dimensions, solution epsilon, root closeness epsilon, and a search time. It is implemented with a genetic algorithm that runs as a client task, whose initial population is strategically selected by the multiple root finder scheduler.

The genetic algorithm task evolves a local population of candidate roots subject to fitness function $-|f(x)|$, a crossover function that produces a child x from parents x_1 and x_2 by means of an arithmetic average, and a mutation function that randomly flips bits in the bitwise representation of x. The multiple root finder scheduler merges the fittest populations that are clustered near a possible root x into a new genetic algorithm task for further refinement, announces a found root x when its fitness is within the solution epsilon, and launches new genetic algorithm tasks until its search time expires. The scheduler discards the results of new searches in the neighborhood of previously found roots, only evolving populations near unseen roots.

3.2 Bitcoin Miner

Bitcoin mining is the act of cryptographically chaining a new block of Bitcoin transactions to the canonical block chain of existing transactions, by manipulating the new block until its SHA256 hash contains a certain number of leading zeros. It is called mining because the miner is rewarded in newly minted Bitcoins for his or her valid solution. A miner forms a new block from uncommitted transactions advertised on the Bitcoin network, a transaction to him or herself for the current mining reward, and some other fields, including a nonce initialized to zero. The miner increments this nonce until the hash of the whole block contains the prerequisite number of leading zeros. If the miner finds this solution before another miner does, he or she advertises the solved block to the network, which will eventually accept it in the canonical block chain, thereby rewarding the miner.

Our Bitcoin miner client task performs the repeated nonce iteration and SHA256 hashing of a potential block, reporting the solution nonce to the scheduler if it is found. Our Bitcoin miner scheduler starts tasks with new block data and a starting nonce value. The miner scheduler partitions the 32-bit nonce value space evenly among the number of available clients, starting each with a different nonce value. If a solution is reported by a task, the miner scheduler submits the solved block to the network, stops all current tasks attempting to solve that block, and starts new tasks with a new block and starting nonces. If a block expires, our minor scheduler stops all current tasks attempting to solve that block, and starts new tasks with a new block and starting nonces.

3.3 MapReduce

MapReduce is a model of computation that processes a list of key-value pairs into a result in a highly parallelizable fashion. The model is based on two phases of computation: $\text{map}(k, v) \rightarrow (k_1, v_1), (k_2, v_2), \ldots$ which accepts a input key-value pair and yield a list of intermediate key-value pairs, and $\text{reduce}(k, (v', v'', \ldots)) \rightarrow r$ which accepts an intermediate key and all values associated with it, and yields a summarized result. The map and reduce phases are defined arbitrarily by the particular application, but must be pure functions, which permits their parallel scheduling among many workers. Map tasks are run concurrently across all input key-values, and then reduce tasks are run concurrently on all intermediate key-values. Reduce tasks results are collected by a coordinator to yield the final result of the computation.

Our implementation of MapReduce includes a map client task, reduce client task, and a MapReduce scheduler. The MapReduce scheduler is responsible spawning map tasks with an input key-value pair (or just an input key, if the corresponding value is large and stored on a shared file system), and for spawning the subsequent reduce tasks with their input as the result of a pending or completed map task. Intermediate results between map and reduce phases are moved by the clients alone, avoiding a data bottleneck at the scheduler. Final results from the reduce tasks are collected at the MapReduce scheduler at the end of the computation. Our MapReduce implementation handles failed tasks gracefully, with the built-in task replication of TaskSprint and a rescheduling policy, to ensure the computation completes despite individual task failures.