Course ID2210
Project Report

Riccardo Reale
Peerialism AB
riccardo.reale@peerialism.com
https://github.com/riccardoreale/id2210-vt14.git

Giovanni Simoni
Peerialism AB
giovanni.simoni@peerialism.com
https://github.com/dacav/id2210-vt14.git

I. Architecture design

A. Base Sparrow

The base system was implemented following the suggested design with the addition of two main components initialized by each simulated Peer:

1) A RmWorker component, which abstracts the Resource Manager from task queue handling and execution.
2) A FailureDetector component, which detects not responding nodes that are subscribed to be monitored. If a monitored node doesn't ACK to alive pings after a certain timeout (3 seconds), the Detector will assume the node to have crashed. For simplicity we assume that on the DataCenter there are no network failures, so if a node is detected as dead, it will stay so.

The required number of CPUs and amount of memory for the execution of a task will be referred as REQUISITES. REQUISITES are allocated by the node executing the task (EXECUTOR) on behalf of the assigner (TASK MASTER). The allocation time is considered as the payload of a task assignment and we assume is not known a priori by a ResourceManager. Therefore the allocation time is used only to simulate the task execution, which consists in allocating the resources (according to REQUISITES), waiting the specified amount of time, and releasing the resources.

We implemented two different behavioral modes. The first implements the best scheduling logic illustrated by the Sparrow paper (Batch + Late Binding), henceforth referred as *Sparrow*, while the second implements an omniscient scheduler for sake of comparison with an optimal assignment strategy, henceforth referred as *Omniscient*. This is obtained directly in the DataCenter scheduler, which, for each task, will calculate the first node to have the requested resources available, and assign it to be executed directly on that node as soon as possible. This operation is, of course, unrealistic since it make use of both updated information about the task queue on each node, and the allocation time of each task.

The workflow for *Sparrow* algorithm is the follow:

1) The task gets issued by the DATACENTER and propagated to a TASK MASTER, which is selected as the closest to the task identifier within a consistent hash table;
2) The TASK MASTER probes a number of his neighbours indicating the REQUISITES.
3) The nodes receiving the probe forwards the request to their RmWorker, which put a place-holder in their own waiting queue, and then signal the TASK MASTER that the probe was received.
4) Every time the RmWorker receives a probe, it selects the first task in the waiting queue (which is FIFO) and, if it has enough resources available to execute it, it temporary allocates the REQUISITES resources and request its ResourceManager to ask for a confirmation from the TASK MASTER.
5) When a TASK MASTER receives a request for confirmation from an EXECUTOR, it send an Assignment to him with the oldest not-yet-assigned task. The assigned task may differ from the task request sent by the probe, as long as it requires less or equal REQUISITES. In case there are no tasks that can be assigned to the available EXECUTOR, it will respond with a Cancel.
6) If an EXECUTOR waiting for a confirmation receives an Assignment, its RmWorker de-allocates the temporary blocked resources, and immediately allocates the assigned task and executes it. This is important since the place-holder task may different from the actually assigned one.
7) If an EXECUTOR waiting for a confirmation receives a Cancel instead, its RmWorker simply de-allocates the temporary blocked resources, and executes 4.
8) When a EXECUTOR RmWorker completes a task execution, it releases the blocked resources, notifies its ResourceManager and executes 4.

Concerning Fault Tolerance, the ResourceManager subscribes to its FailureDetector each node it probes and check that every probe is received. If a node is detected as dead, the TASK MASTER will restart the probe process for each task that was running on the dead node. Similarly the RmWorker of an EXECUTOR subscribes each TASK MASTER. If a TASK MASTER is detected as dead, the RmWorker will cancel all his tasks.

B. Improved Sparrow

The improved version of Sparrow consists in four main algorithm modifications:

1) **Self Assignment** - A ResourceManager receiving a Task from the DataCenter will directly execute it if its RmWorker has enough resources immediately available.
2) **Probe Propagation** - A ResourceManager receiving a probe can decide to deposit it on its RmWorker or

propagating it to another neighbour. The probe will be locally deposited if its task can be executed immediately or it has been propagated more than a fixed number of time (usually 5).

3) **Next Hop Selection** - When probing or propagating a probe, the peer selected from the neighbours can be chosen randomly (Random Walks) or in a greedy way (Greedy Search), based on the current knowledge of his resources availability provided by Cyclon We used a SoftMax algorithm with configurable temperature.
4) **Gradient View** - Instead of using the the neighbors view provided by Cyclon, the RmWorker can use the view provided by Gradient, implemented using a Gradient Ranking function on top of TMan, based on the available node resources. We used a greedy ranking function, while the utility function is very basic and defined as:

$$U_p(n) = CPU_p(n) - Queue_p(n) \quad (1)$$

Where $CPU_p(n)$ means the available CPU of the node n according to the value cached by node p, and $Queue_p(n)$ the length of its waiting queue. The reason of this choice will be clarified in the Evaluation.

II. Experimental evaluation

A. Setup

To evaluate our improvement over Sparrow we selected two main scenarios:

- A load scenario where 50 homogeneous nodes executes 1000 tasks, each requesting 2 cpus and 2Gb of memory and a fixed execution time of 10 seconds. The load simply determines the inter-arrival time of the tasks, based on the number of total cpu in the DataCenter (the dominant resource) and the processing time.
- A random scenario instead consists in using a variable number of heterogeneous nodes, executing 500 tasks with different requirements in terms of cpu and time, which arrive at a rate of half second each.

Both scenarios are generated using the CPU as dominant resource, to allow us to easily control the load and greedy ranking functions, both for Greedy Search and Gradient.

We added and evaluated the different algorithm improvements adding them to Sparrow base one to obtain three different configurations:

1) **Random Walk** uses Self Assignment and Probe Propagation, with random Next Hop Selection (Temperature set to 100), giving us Random Walks for each probe.
2) **Greedy Search** adds Greedy Next Hop Selection.
3) **Gradient Search** uses Greedy Next Hop Selection on top of Gradient View.

As main measures we use, as suggested, the average queue time of each task and the 99th percentile. Each result is average over 5 runs with different seed.

B. Results in Load Scenario

Figure 1 shows the base implementation of Sparrow in the Load scenario, for different number of probes used. The results are consistent with the one presented in the Sparrow work.

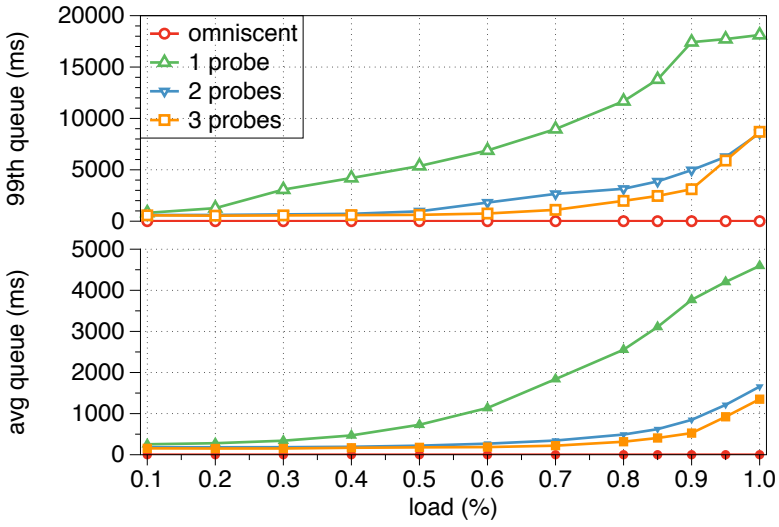

Fig. 1. Base Sparrow algorithm using different load and number of probes

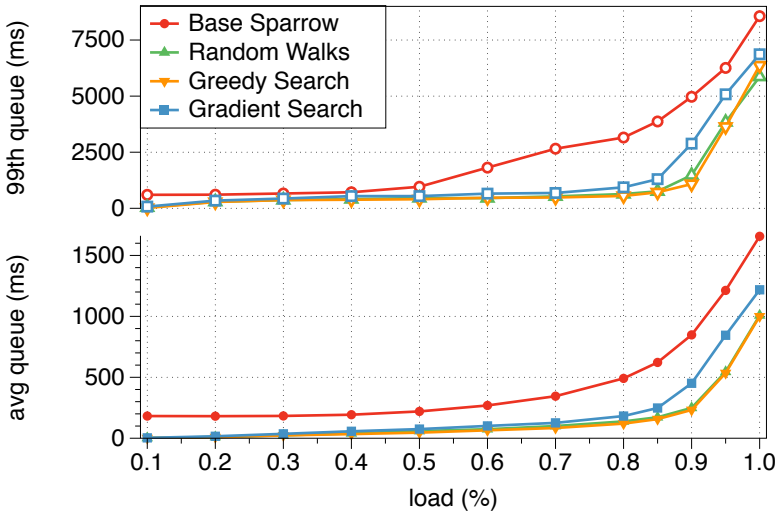

Fig. 2. Comparison between Base Sparrow Algorithm and its enhancements

Figure 2 instead shows the results comparison between the Base Sparrow algorithm (2 probes) and the improved configurations.

The comparison shows how the Self Assignment decreases the average queue time, since in most cases the task won't be propagated at all but immediately executed on the Task Master.

The Probe Propagation instead decreases dramatically the 99th percentile queue time, especially with loads between 50% and 85%, and similarly on all the configurations. Above 85%, all configurations experience an increase in both average and 99th percentile. The last configuration, using Gradient, performs slightly worse than the others. At higher loads probably Cyclon is not fast enough to provide very fresh samples of the neighbour resources. Using a Gradient Search on top of Random View instead doesn't improve significantly neither average or 99th percentile.

The Gradient Ranking function is not too useful when the load increases too much since the use of Late Binding provides a poor correlation between waiting queue length and actual load, since any of the place-holder could get a Cancel on confirmation request.

C. Results in Random Scenario

Finally we show the results in the Random Scenario