Cloud Scheduling

What are the considerations?

- What is scheduled?
  - Map-Reduce / Dryad model:
    - Complete parallel computation jobs
    - Short term
  - Web services:
    - Numbers of machine to allocate to a service
    - Long term
- What are the goals?
  - Performance
  - Efficiency (low bandwidth on expensive links)
  - Fairness of multiple jobs
  - Low energy
  - Low temperature

Quincy

- Data-intensive compute clusters have a range of job lengths, but most < 30 minutes
- Users want fairness:
  - Nobody monopolizes all machines
- Systems needs locality
  - Minimize bandwidth needs
- These conflict:
  - Waiting for machine with your data may delay you & reduce fairness

Motivation

- Fairness:
  - Existing dryad scheduler unfair [greedy approach].
  - Subsequent small jobs waiting for a large job to finish.
- Data Locality:
  - HPC jobs fetch data from a SAN, no need for co-location of data and computation.
  - Data intensive workloads have storage attached to computers.
  - Scheduling tasks near data improves performance.

Fair Sharing

- Job X takes t seconds when it runs exclusively on a cluster .
- X should take no more than Jt seconds when cluster has J concurrent jobs.
- Formally, for N computers and J jobs, each job should get at-least N/J computers.

Fine Grain Resource Sharing

- For MPI jobs, coarse grain scheduling
  - Devote a fixed set of computers for a particular job
  - Static allocation, rarely change the allocation
  - Killing/moving expensive: lots of state, lots of communicatoin
- For data intensive jobs (map-reduce, dryad)
  - We need fine grain resource sharing
    - multiplex all computers in the cluster between all jobs
  - Large datasets attached to each computer
  - Independent tasks (less costly to kill a task and restart)

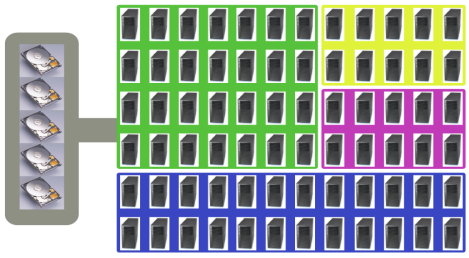
How share?

- How do you achieve locality?
  - Wait for the computers with your data to be available
  - This leads to unfairness when long-running jobs are using them
- How do you achieve fair share?
  - Kill some jobs (preemption)
  - Run jobs earlier on non-optimal computers (remote data access)
  - Both of these have costs …

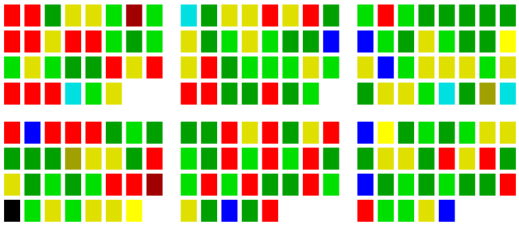
Computation model

- A job = a computation made of many tasks/workers
- A single scheduler maintains a job queue of concurrent jobs
  - The master for a job submits a list of workers to the scheduler to be run
  - Finishing a worker may trigger the master to submit more jobs
  - Each worker is annotated with location of its data + size
    - Schedule can compute transfer cost from different locations
    - Preferred locations = computers that have 10% of data or more
    - Preferred racks = racks with > 10% of input data
- Scheduler places jobs from the queue and can kill them to reclaim the machine

Example of Coarse Grain Sharing



Example of Fine Grain Sharing

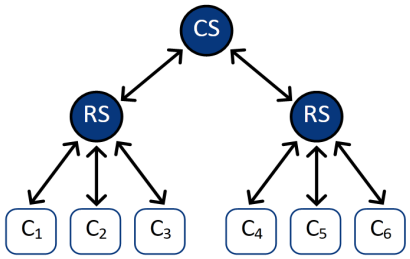


QUESTION: Why fine grained? More flexibility in creating jobs

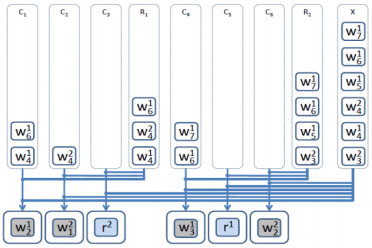
Goals of Quincy

- Fair sharing with locality.
- N computers, J jobs (independent of # of workers),
  - Each job gets at-least N/J computers
  - With data locality
    - place tasks near data to avoid network bottlenecks
  - Feels like a multi-constrained optimization problem with trade-offs!
  - Joint optimization of fairness and data locality
  - These objectives might be at odds!

Cluster Architecture



Baseline: Queue Based Scheduler



Job assigned to preferred computer, rack, and X

Each computer (c) and rack (r) have a queue. Idle computers pull from queue Ci then from rack queue Rj then common queue X

Greedy Sched Problems: fairness for new jobs

- Biased towards jobs with lots of tasks
  - Can fill up all the queues, new tasks don't start
- Long latency on loaded cluster
  - Already tasks in all the queues from other jobs
- Solution 1: block jobs using more than fair share of machines
  - Prevents it from submitting new workers when it already has its share
  - But tasks already in the system can cause unfairness…
- Solution 2: preemption
  - Kill some tasks if a job has more than its share of workers

Quincy Idea

- Compute cost of a scheduling decision (delaying, running on wrong machine, or killing a task)
- Compute optimal assignment from these costs
- Express problem as a flow network
  - Nodes in the network are racks, machines, tasks
  - Add edges from machines to queues where they are scheduled
  - Adjust weights & capacity of network to account for data locality

Flow Based Scheduler = Quincy

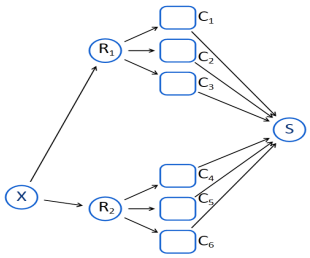
- Main Idea: [Matching = Scheduling]
  - Construct a graph based on scheduling constraints, and cluster architecture.
  - Assign costs to each matching.
  - Finding a min cost flow on the graph is equivalent to finding a feasible schedule.
  - Each task is either scheduled on a computer or it remains unscheduled.
  - Fairness constrains number of tasks scheduled for each job.

Graph Construction

- Flow networks:
  - Each link has a capacity
  - Each node has a supply
  - Each node is balanced: flow in + supply= flow out
- Start with jobs
  - Each one has a "token" of work
  - Each computer can do a "token" of work (no sharing)
  - Goal: assign flow of jobs to machines
- Add weights to reflect scheduling choices
- NOTE: graph evolves over time

Graph Construction

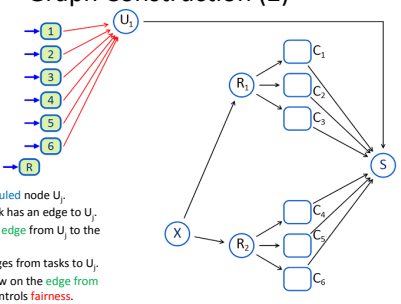
- Start with a directed graph representation of the cluster architecture.



Add jobs

- Initially unscheduled, so connected there
- Each job has supply 1
- Cost on link to U is the cost of leaving a job unscheduled
  - E.g. could be delayed until a better option available
  - Can increase over time to add pressure to schedule
- Supply of U ->sink = number of jobs that can be left unscheduled

Graph Construction (2)

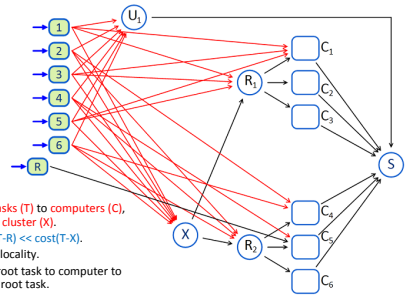


- Add an unscheduled node U_j.
- Each worker task has an edge to U_j.
- There is a single edge from U_j to the sink.
- High cost on edges from tasks to U_j.
- The cost and flow on the edge from U_j to the sink controls fairness.
- Fairness controlled by adjusting the number of tasks allowed for each job

Connect tasks to preferred locations

- Connect to preferred racks and computers
- Weight on edges = communication cost to that place
- Connect tasks to root X with default cross-rack cost
  - Avoids connecting tasks to all computers
- Connect tasks to preferred racks
  - Cost is cross-node but intra-rack communication
  - Connect tasks to preferred nodes
  - If task is running somewhere, connect there

Graph Construction (3)



- Add edges from tasks (T) to computers (C), racks (R), and the cluster (X).
- cost(T-C) << cost(T-R) << cost(T-X).
- Control over data locality.
- 0 cost edge from root task to computer to avoid preempting root task.

Add killing links

- When task starts on a computer C, increase cost to all nodes other than C reflecting cost of killing/moving the job

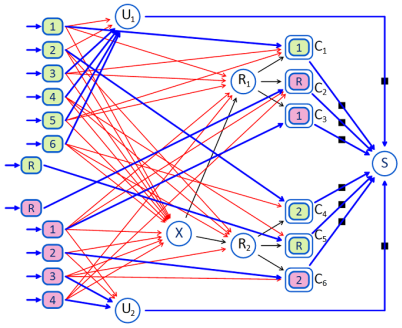
Adding fairness

- Change the number of jobs that can be unscheduled
  - If 0: have preemption (must take a computer from someone else)
  - If 0 but remove links to other nodes once job is scheduled: fair share w/o preemption
  - If set to n-1 tasks of a job -> unfair sharing
- Control costs of leaving a task unscheduled
  - Initially zero, can raise over time

Quincy Assumptions

- Single job per node
  - No worries about different constraints (e.g., network, memory)
- Independent jobs
  - No constraints that two tasks must be on same node
  - Dryad handles this before creating tasks
- Global uniform cost measure
  - E.g. Quincy assumes that the cost of preempting a running job can be expressed in the same units as the cost of data transfer.
  - Allows for single cost metric on a flow

Final Graph



How to use Graph

- Run a solver to find the min flow solution
  - Fairly fast (6ms for 243 nodes)
- When to run?
  - Whenever set of jobs changes

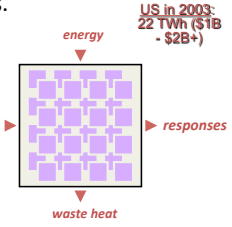
Energy-aware Scheduling

- Problem:
  - Idle machines use more than zero power
  - Reduce data center efficiency
- Solution:
  - Consolidate workloads onto fewer machines

Managing Energy and Server Resources

- Key idea: a *hosting center OS* maintains the balance of requests and responses, energy inputs, and thermal outputs.
  1. Adaptively provision server resources to match request load.
  2. Provision server resources for energy efficiency.
  3. Degrade service on power/cooling failures.

     Power/cooling *"browndown"*
     Dynamic thermal management
     [Brooks]



Contributions

- Architecture/prototype for *adaptive provisioning* of server resources in Internet server clusters (*Muse*)
  - Software feedback
  - Reconfigurable request redirection
  - Addresses a key challenge for hosting automation
- Foundation for energy management in hosting centers
  - 25% - 75% energy savings
  - Degrade rationally ("gracefully") under constraint (e.g., browndown)
- Simple "economic" resource allocation
  - Continuous *utility functions*: customers "pay" for performance.
  - Balance service quality and resource usage.

Utilization Targets

μ_i = allocated server resource for service i

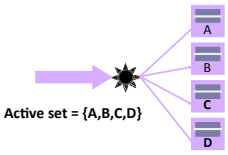
ρ_i = utilization of μ_i at i's current load λ_i

ρ_{target} = configurable target level for ρ_i
Leave headroom for load spikes.

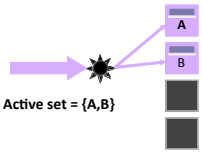
$\rho_i > \rho_{target}$: service i is *underprovisioned*

$\rho_i < \rho_{target}$: service i is *overprovisioned*

Energy vs. Service Quality

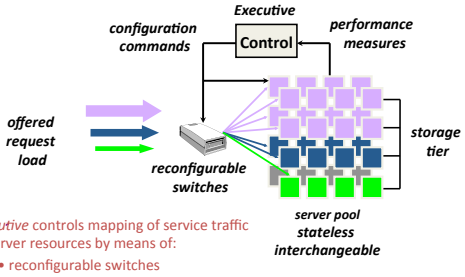


$\rho_i < \rho_{target}$
- Low latency



$\rho_i = \rho_{target}$
- Meets quality goals
- Saves energy

Muse Architecture



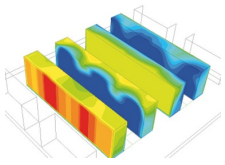
Executive controls mapping of service traffic to server resources by means of:
- reconfigurable switches
- scheduler controls (shares)

Energy-Conscious Provisioning

- Light load: concentrate traffic on a minimal set of servers.
  - Step down surplus servers to a low-power state.
    - *APM* and *ACPI*
  - Activate surplus servers on demand.
    - *Wake-On-LAN*
- Browndown: can provision for a specified energy target.

Temperature-aware Scheduling

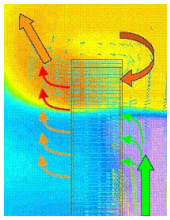
- Challenge: data centers have cold aisles (air through floor) and hot aisles (air sucked out through machines)
- Variation in temperatures can lead to extra expense air conditioning
  - Need to draw off all heat
  - Cold air cools better than hot air
  - Hot air is cheaper to produce than cold air
  - Need to prevent all areas from getting to hot



Thermal issues in dense computer rooms

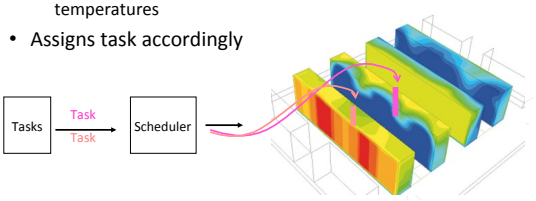
(i.e. Data centers, Computer Clusters, Data warehouses)

- **Heat recirculation**
  - Hot air from the equipment air *outlets* is fed back to the equipment air *inlets*
- **Hot spots**
  - Effect of Heat Recirculation
  - Areas in the data center with alarmingly high temperature
- **Consequence**
  - Cooling has to be set very low to have *all* inlet temperatures in safe operating range



Functional model of scheduling

- Tasks arrive at the data center
- Scheduler figures out the best placement
  - Placement that has minimal impact on peak inlet temperatures
- Assigns task accordingly



Conceptual overview of thermal-aware task placement

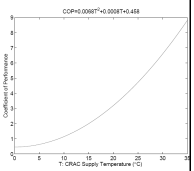
Task placement determines temperature distribution

Peak air inlet temperature determines upper bound to CRAC temperature setting

Temperature distribution determines the equipment peak air inlet temperature

CRAC temperature setting determines it's efficiency (Coefficient of Performance)

The lower the peak inlet temperature the higher the CRAC efficiency



Coefficient of Performance
(source: HP)

bottom line

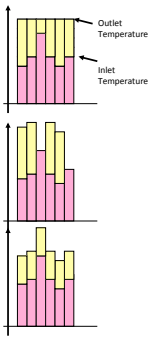
There is a task placement that maximizes cooling efficiency. ***Find it!***

Temperature Scheduling

- Energy usage = heat production on a server
  - Less workload, less heat
- Place jobs according to availability of cheap cooling
  - Fewer jobs where harder to cool
  - More jobs where cooling more efficient
    - E.g. those with coldest air coming in

Contrasted scheduling approaches

- Uniform Outlet Profile (UOP)
  - Assigning tasks in a way that tries to achieve uniform outlet temperature distribution
  - Assigning more task to nodes with low inlet temperature (water filling process)
- Minimum computing energy
  - Assigning tasks in a way that keeps the number of active (power-on) chassis as few as possible
  - Server with coolest inlet temperature first
- Uniform Task (UT)
  - Assigning all chassis the same amount of tasks (power consumptions)
  - All nodes experience the same power consumption and temperature rise



Delay Scheduling

- Similar goal to Quincy
  - Shared clusters with data stored on nodes
  - Sequence of jobs accessing data with variable needs (e.g. 7500 jobs) at low latency (e.g. jobs average ~90s)
- Goal: max-min fairness
  - Minimum rate of maximum job or maximum rate for minimum job (reduce outliers)
    - Requires reallocating resources as jobs come & go
- Goal: data locality
  - Schedule jobs on nodes that have their data

Problem

- When new job arrives and needs to run, do you:
  - Kill some jobs to make space
  - Wait for some jobs to finish?
- Basic scheduler: puts jobs in queue
  - When an node needs work, request task from queue
  - Sort by jobs with fewest tasks
- With short jobs, better to wait (avoids wasted work)
- If you wait just for next available machines, small jobs have poor locality
  - Unlikely to get a machine with their data, leads to more cross-node/rack communication
- With larger jobs, have *sticky slots*
  - A job that is finishing will tend to get rescheduled on the node where it was just running (because that space was just idled)
    - Without queues…
    - Means bad assignments tend to persist

Delay Scheduling

- Key idea: better to wait for the right slow when jobs are fairly short then take the next available one
  - When machine is free & needs a task, look for first task that has data on that machine
  - If a job is skipped more than SkipCount times, run it somewhere else
    - If machines freed frequently and not run, then quickly runs elsewhere
    - If machines freed rarely, takes longer to move somewhere else
    - E.g. not clock based
- Challenges:
  - What if one machine has lots of popular data?
  - What if there are long running jobs on the machines with your data?
  - **ANSWER: get to run if sitting for too long**

Comparison to Quincy

- Allows multiple jobs per machine (e.g. 1 per core)
  - Chance of being stuck is much lower, so not need preemption
- Uses rate of machines being freed up to make decision
  - When considering how long to skip a task before placing it elsewhere

Overbooking

- Assume each job needs local resources
  - CPU, memory, network disk
- Want to allow multiple jobs per machine for efficiency
- How much can you overbook
  - E.g. assign jobs such that worse cases don't fit?

Overbooking approach

- First: characterize job requirements
  - Measure resource usage over time when standalone
  - Determine requirements from trace
  - Determine distribution of usage from trace
    - E.g. in a period, what is probability of using x% of the resource?
  - Determine tolerance for failure
    - Is statistically possible

Overbooking (2)

- Second: assign jobs to machines
  - Case 1: worst case behavior of all jobs fits
    - Easy
  - Case 2: worst case doesn't fit
    - First, make sure common-case usage fits for all resources
    - Second: ensure probability of overbooking a resource fits requirements
      - Look at probability for all jobs to use the resource, combine to figure probability of it being oversubscribed

Overbooking (3)

- Third: assign jobs to machines
  - Make bipartite graph of jobs to machines they can run on
  - Start with most restricted job (e.g. first one in figure), schedule it first
  - If multiple choices, use (for multiple resources)
    - best fit - leaves open more capacity on other machines
    - Worst fit – lots of idle resources locally