Resource Management Part I

Eran Smadar, Spring 2015
Outline

• Introduction – large-scale compute
  • Definition, terminology, examples, typical environment
  • Roles of the scheduler

• Proportional-share scheduling
  • Motivation (org-tree model)
  • Challenges
    1. Measuring resource consumption
    2. Ensuring fast ramp-up
    3. Considering history
Introduction
What is large scale computation?

• **Batch processing** is the execution of a series of programs ("jobs") on a computer without manual intervention

• **Parallel computing** is a form of computation in which many calculations are carried out simultaneously (usually run in batch)

• **Large scale computation** is a model in which the batch work is carried out on a large cluster (or a parallel machine)
  • This work is coordinated by an entity called the ‘resource management system’ (RMS) or simply ‘the scheduler’
  • The main characteristic in such a model is parallelism: many jobs (or tasks), by many users, execute in parallel and share the cluster
  • Users wait for their jobs to complete, receive an answer, and possibly submit more jobs
Why do we need this?

• Get the work done faster
  • e.g., 10K simulations that execute for one hour each, will take more than a year to complete on an average laptop and potentially one hour if done in parallel
Examples

• Hadoop map-reduce
  • Each job translated into multiple maps and reduce ‘tasks’ executed in parallel

• Weather forecasting
  • Often encompasses true parallel computations, e.g., MPI

• Validating chip designs
  • Numerous independent simulations executed in parallel

• Bitcoin mining
  • Uses special HW to accelerate the search
Terminology

• “Job” is the basic unit of computation
  • This is what the user ‘submits’ to the scheduler
  • Serial vs. parallel jobs (MPI)

• Job may be composed of multiple tasks
  • Map-reduce tasks in Hadoop
  • MPI tasks in MPI

• Each job has an “owner”
  • The user who submitted the job

• Each job comes with requirements
  • Resources (memory, cores, disk space, etc.)
  • Operating-System to execute on
  • Licenses, etc.
Typical environment

• Hundreds to tens-of-thousands of users
  • Each submitting multiple jobs

• Diverse jobs requirements
  • Memory, Cores, OS, etc.

• Different Service Level Agreements (SLA)
  • Priority, e.g., ‘I want my jobs to run first’
  • Share, e.g., ‘I paid, so I want 20% of the resources’

• Hundreds to tens-of-thousands of servers
  • Each capable of executing multiple jobs

• Different geographic locations
  • Meta-scheduling (RM part III)
Typical environment cont.

- Resource capacity is limited
  - Driven by amount of servers and configuration
  - Governed by datacenter space, cooling, and power budgets (facility lecture)
- Demand often exceeds the capacity
  - Jobs must be queued and only then scheduled for execution (when resource becomes available)
- This is what makes the problem interesting 😊
  - Which job(s) to schedule next?
  - How to minimize fragmentation?
Problem is generic

• Scheduling Virtual Machines (VMs), e.g. Amazon
• Scheduling map-reduce jobs e.g., Hadoop
• Scheduling parallel jobs, e.g., Slurm
• Scheduling chip simulations e.g., Intel
• Etc.

Logos of Amazon, Hadoop, Blue Gene & Intel may be trademarks of respective companies
Running efficient batch environment

• There are many aspects related to running an efficient batch environment
  1. Job scheduling
  2. Resource allocation (matching)
  3. System uptime
  4. Job management capabilities
  5. Environment management capabilities
  6. Informative indicators
  7. Monitoring systems
  8. Predictive capabilities
  9. More ...

Challenges in Modern Data Centers Management, Spring 2015
Running efficient batch environment

• There are many aspects related to running an efficient batch environment
  1. Job scheduling
  2. Resource allocation (matching)
  3. System uptime
  4. Job management capabilities
  5. Environment management capabilities
  6. Informative indicators
  7. Monitoring systems
  8. Predictive capabilities
  9. More ...

In this lecture we’ll focus on challenges related to Job scheduling
Next lecture will focus on challenges related to Resource Management
Batch job schedulers

• Software that provides the facility to queue jobs and schedule them on available resources according to some pre-defined policy
  • LSF (Batch scheduler), SLURM (HPC schedule), Condor (Grid scheduler), etc.

• Two actions by the scheduler
  1. Job scheduling – this lecture
     • Selecting the next job(s) to execute
  2. Resource matching (allocation) – next lecture
     • Matching the job(s) with available resources

• Both are interdependent & must be done fast (hundreds of jobs per second in large installations)
Bad scheduling – can you bear it?

Credits to Ferdinand Lutz. Taken from YouTube - http://www.youtube.com/watch?v=IPxBKxU8GIQ (used for education purposes only)

Challenges in Modern Data Centers Management, Spring 2015
Proportional-share scheduling
Motivation (org-tree model)

- Big organizations are hierarchical
  - Business units (Virtual Organizations or VO), Projects, Users, etc.
- Each VO uses its budget to buy equipment (servers)
  - Also decides which equipment goes to which project(s), user(s), etc.
- Equipment joins a shared pool of resources
  - Q: Why would a VO share its equipment?
- Goal is to ensure that each entity (VO, projects, users) gets its **proportional-share** of the resources
  - This is the “SLA” the scheduler needs to guarantee
- Other models exist e.g., pay-per-use
  - Different SLA definitions

**Target tree**

- Data structure corresponding to the org-tree, which defines for each node in the tree the share of the resources it **should get**
  - Shares are normalized to ease calculations
  - Siblings shares always sum to 1.0

- Defining the shares can be challenging
  - Assume VO1 bought 8-cores X 64GB servers, and VO2 4-cores X 128GB servers
    - Q: Which share should each get?
  - Are cores & memory only parameters to consider?
Online proportional-share algorithm

• Uses a greedy approach (one job at-a-time)
  1. Identify the most “needy” leaf in the tree (project or user)
  2. Start executing a job from that leaf (resource allocation – RM part II)
  3. Return to step #1

Definitions
• VO = Virtual Organization (==business unit)
• P= Project
• U=User
• LQ = Local queue (in case we have multiple sites)
Identifying the most “needy” leaf – Step 1

- For each node calculate its **use of the resources**
  - Starting with the leafs, each node is the sum of its descendants
  - Normalize the result (normalized usage tree)

- Calculating usage can be challenging
  - For now assume we simply count the # of running jobs at the leafs
  - No history considered

---

**Raw Usage Tree**

- RS
  - VO1
    - P1
      - U2
      - U3
      - U4
  - LQ
    - U1
  - VO2
    - P2
      - U1
      - U3
      - U4

**Normalized Usage Tree**

- RS
  - VO1
    - P1
      - U2
      - U3
      - U4
  - LQ
    - U1
  - VO2
    - P2
      - U1
      - U3
      - U4

---

Challenges in Modern Data Centers Management, Spring 2015
Identifying the most “needy” leaf – Step 2

- Calculate the “diff” between the shares in the **target tree** and the shares in the **normalized usage tree**
  - Should-get minus Getting-now
- Result called “**Priority tree**”

Identifying the most “needy” leaf – **Step 3**

- Traverse the priority tree from the root along the path with the highest diff
  - Stop when reaching the most “needy” leaf
- Start executing a job from that leaf
  - Return to step 1

Challenges in Modern Data Centers Management, Spring 2015
Online proportional-share: **Summary**

• Sharing is enforced in a **top-down** manner
  1. VO (BU) are prioritized (more important) over projects
  2. Projects are prioritized over users, etc.
Challenges

1. Calculating resource usage
2. Ensuring fast ramp-up
3. Considering past usage (history)
Challenge #1: Calculating resource usage

• **Idea:** If we think of the resources that are allocated to a VO as money, and the jobs running on these resources as spending that money, then some jobs can be thought of as being more “expensive” than others.
  - Intuitively, jobs that consume more resources should “pay” more.

• **How to calculate resources usage?**
  - Derivatives:
    1. Calculating the target tree
    2. Calculating the usage tree
    3. Calculating system utilization
Single-dimension: Count jobs cores (or memory)

- Count cores used by the jobs at the leafs and sum towards the root
  - Also, allocate resources solely based on the cores requested by the jobs (RM part II)

- Pros 😊
  - Simple (allocate and count cores)
  - Result (calculated usage) reflects actual resource usage (# of cores used)

- Cons 😞
  - Single dimension (considers a single resource)
  - In reality jobs request both cores and memory
    - Also, servers come with different “shapes”
  - Which job consume more resources?
    - 1-core X 4GB or 3-cores X 1GB
Multiple dimensions: Count job slots

- **Job slot** is a **fixed-size partition** of the underlying servers resources
  - E.g., 1 core X 8GB of memory

- Count jobs slots used by the jobs at the leafs and sum towards the root
  - Jobs request one or more **job slots** in order to execute
  - Allocate resources in units of jobs slots (**RM part II**)  

- **Pros ☺**
  - Simple (allocate and count job slots)
  - Accounts for multiple dimensions

- **Cons ☹**
  - Calculated usage does not reflect actual resource use
  - What if a job only needs 5 GB of memory?
    - Severe over and under-utilization of resources
Multiple dimensions: Dominant Resource Fairness (DRF)

- [https://www.cs.berkeley.edu/~alig/papers/drf.pdf](https://www.cs.berkeley.edu/~alig/papers/drf.pdf)
- Idea: focus on the bottleneck resources
  - If user A’s jobs are “heavy” on cores, and user B’s are “heavy” on memory, calculate user A’s share relative to the cores, and user B’s share relative to memory
  - Balance the shares based on the dominant resources (example in next slide)
- Pros 😊
  - Accounts for multiple dimensions
  - Reflects actual resource usage (as opposed to job slots)
- Cons 😞
  - Usage becomes complicated
    - Resource consumption for the same job can count differently depending on the servers, making it hard to explain to the users (less intuitive)
DRF – Example

- Server has 9 cores X 18GB memory
- User A has 1-core X 4GB memory jobs
  - Each job consumes 1/9 of the cores, and 2/9 of memory ➔ memory is the dominant resource for user A
- User B has 3-core X 1GB memory jobs
  - Each job consumes 1/3 of the cores, and 1/18 of memory ➔ cores is the dominant resource for user B
- Allocating 3 jobs from user A and 2 from user B results in equal sharing of the dominant resources
  - 2/3 of the server memory for user A (memory = user A’s dominant resource)
  - 2/3 of the server cores for user B (cores = user B’s dominant resource)
Best practices

• DRF was the best option so far
  • Accounts for multiple dimensions (both in jobs and servers)
  • Accurate allocation of resources (as opposed to job slots)

• But usage became complicated (less intuitive)
  • Big issue in production environment involving users

• A possible tradeoff....
  • Accurately allocate resources (using both cores and memory)
    • Avoiding under or overutilization of resources
  • When calculating usage count only cores
    • Good for workload in which cores is the dominant resource
## Challenge #1: Calculating resource usage

### Summary

<table>
<thead>
<tr>
<th>Solution</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Allocate and count cores (or memory)</td>
<td>• Simple</td>
<td>• Single dimension (does not reflect real world scenario)</td>
</tr>
<tr>
<td>2. Allocate and count jobs slots</td>
<td>• Simple</td>
<td>• Allocation in units of jobs slots does not reflect actual resource usage of the jobs, resulting in severe under and over utilization of resources</td>
</tr>
<tr>
<td>3. Dominant Resource Fairness (DRF)</td>
<td>• Accounts for multiple dimensions (both in cores and servers)</td>
<td>• Complicated usage (less intuitive)</td>
</tr>
</tbody>
</table>
Challenges #2: Ensuring fast ramp-up

• Proportional-share works fine as long as there is **constant steam of jobs**
  • Each leaf has **at least** one waiting job

• What if V02 doesn’t have jobs to send?
  • Its share (0.3) gets distributed among V01 and LQ

• When P3 becomes active it has **no resources available**
  • Must wait for jobs from V01 and LQ to complete to free resources

• This is called the **ramp-up problem**
  • Time that elapses between when a leaf becomes active and when it gets its promised share of the resources
Ramp-up: Suspending or restarting jobs

• Suspend or restart jobs from other leafs

• Pros 😊
  • “needy” leaf immediately gets its share (immediate ramp-up)
  • Keeps the system utilized

• Cons 😞
  • Not always feasible (not all jobs may be suspended or restarted)
  • Impacts **effective utilization** (in case of restarts)

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Suspension</th>
<th>Restart</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success rate</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>Job turn around time</td>
<td>Depends on the suspended job state and the other jobs on the server</td>
<td></td>
</tr>
<tr>
<td>Resource release</td>
<td>Not completely, e.g. disk space, virtual memory</td>
<td>completely</td>
</tr>
</tbody>
</table>

Challenges in Modern Data Centers Management, Spring 2015
Ramp-up: Using logical buffers

• Placing a “max-limit” on different nodes in the tree e.g., max-running
  • Limits the # of running jobs from these nodes (effectively reserving buffers for ramp)

• Pros 😊
  • Good compromise: enables to improve utilization (up to the limits)
  • Flexible: limits can be tuned to ensure fast ramp till reaching a configured threshold

• Cons 😞
  • Manual tuning (usually)
  • Not ideal utilization (buffers remain idle)

• Note:
  • Usually best HW is used for high priority activities
    • We want to keep the best resources utilized
Ramp-up: Using physical buffers

- Reserving dedicated servers to the activity (node)
  - Effectively reserving buffers for ramp-up if needed
- Pros 😊
  - Fixed resources – we can pick the best HW
- Cons 😞
  - Break the sharing model, hurts utilization
- Q: How can we utilize the buffered resources?
  - E.g., Prediction?

Challenges in Modern Data Centers Management, Spring 2015 34
Challenge #2: Ensuring fast ramp-up

Summary

<table>
<thead>
<tr>
<th>Solution</th>
<th>Pros</th>
<th>cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Don’t use the resources</td>
<td>• Avoids the problem</td>
<td>• Huge waste of resources</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Does not encourage sharing</td>
</tr>
<tr>
<td>• Suspend or restart jobs</td>
<td>• Ensures fast ramp-up</td>
<td>• Suspension not always feasible</td>
</tr>
<tr>
<td></td>
<td>• Keeping system utilized</td>
<td>• Can significantly impact jobs turn around times</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>Hurts effective utilization</strong></td>
</tr>
<tr>
<td>• Using logical buffers (max-</td>
<td>• Ensures fast ramp-up</td>
<td>• Hurts the utilization</td>
</tr>
<tr>
<td>limits)</td>
<td>• Compromise between utilization</td>
<td></td>
</tr>
<tr>
<td></td>
<td>and success rate</td>
<td></td>
</tr>
<tr>
<td>• Using physical buffers</td>
<td>• Ensures fast ramp-up</td>
<td>• Hurts utilization, usually, of the best resources</td>
</tr>
<tr>
<td></td>
<td>• Can designate specific hardware</td>
<td></td>
</tr>
</tbody>
</table>

- We should pick the metric according to the workload and the information that we have on the jobs
- For example: if we want to reduce the average turn around time of a jobs, what would be the best metric?
  - Don’t use the resource? Hurts utilization → increase wait time
  - Restart jobs? depends on the preemptive and preempt-able job
  - ...
Challenge #3: Considering past usage (history)

• Two ways to calculate resource use:

  1. History does **not** matter (memory-less)
     • Resources should be shared solely based on current demand
       • You didn’t use? It’s your problem…

  2. History **does** matter (memory based)
     • Resources should be shared considering past resource usage
       • You didn’t use? – You **may be** entitled to compensation
Memory-less vs. Memory-based

- Assume 2 activities: A & B, both sharing a cluster. Each activity paid for certain share of the resources, e.g. 70% for A, and 30% for B

Memory less – history does not matter

Memory based – history does matter
Example use cases

• Memory-less: constant demand, while variety for demand is much lower than the amount of resources

• Memory-based: the amount of activities to serve is larger than the amount of resources in the system
Memory-based implementation

- Sliding-window approach
  - Use sliding window to record historical resource usage (per activity/node)
  - Consider the information in the window to decide which activity to schedule next
## Challenge #3: Considering History

### Summary

<table>
<thead>
<tr>
<th></th>
<th>Memory-Less</th>
<th>Memory-Based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation</td>
<td>Easy</td>
<td>More complex</td>
</tr>
<tr>
<td>Utilization</td>
<td>Higher</td>
<td>Lower</td>
</tr>
<tr>
<td></td>
<td>Utilizing empty cycles</td>
<td>Pay for every cycle</td>
</tr>
<tr>
<td>Usage management</td>
<td>Easy</td>
<td>Hard</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Requires planning</td>
</tr>
<tr>
<td>Visibility &amp; administration</td>
<td>Easy</td>
<td>Hard</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Difficult to understand/explain why jobs are waiting</td>
</tr>
</tbody>
</table>

- Memory-based is good at **handling peaks**: as it compensates users/projects which did not utilize their share by giving them higher share for a limited duration of time.
References

• FSGrid

• DRF
  • https://www.cs.berkeley.edu/~alig/papers/drf.pdf
Backup