Resource Allocation in the Grid Using Reinforcement Learning

Presented by: Xiaolong Jin

Based on a paper with the same name authored by A. Galstyan, K. Czajkowski, and K. Lerman, available at: http://www.isi.edu/~Elerman/papers/P-243.pdf

Introduction

- Grid computing: enabling sharing of numerous computing resources over the network
- Virtual organizations (VOs): associating heterogeneous users and resources
- Resource allocation: mapping users' jobs to specific resources in order to optimize some utility metric
  - Situation: tens of thousands of users and thousands of resources
  - Requirements on allocation mechanism:
    - Highly scalable
    - Robust to localized failures: dynamical arrival and departure of users and resources; communication
- Objective: study resource allocation from a learning and adaptation perspective

Grid Scheduling Issues

- Observations:
  - Due to decentralized nature of the Grid:
    - Different portion of the Grid may use different resource allocation strategies
    - A centralized allocation scheduler is not feasible
    - Users have limited real-time environmental knowledge, including resource status information
  - *Understanding of the effects of different resource allocation mechanisms on global system behavior will influence architectural decisions as well as the policies chosen within federated VOs.*
- Solution:
  - Decentralized scheduling mechanism
  - Not depend on the availability of the current global knowledge

The Model:

Resource Providers

- Local scheduling of Computational tasks:
  - Resource characterization:
    - Number and speed of the processors available
    - System memory
    - Storage space
  - Multiple jobs can run simultaneously in the system
  - Different scheduling policies:
    - FCFS (First come first serve)
    - LJF (Long job first)
- Authors' setting: Simplified resource representation and local scheduler
  - Resource characterization: Computation power, i.e., CPU time/unit job
  - A single job run at each given time
  - Jobs are prioritized according to its arrival time: FCFS

The Model:

Users

- User roles:
  - Individual agent: generate jobs, search a resource for each job ✓
  - Resource broker: search resources for jobs on behalf corresponding users
- Users modeling:
  - Feature: Heterogeneous selfish agents
  - Goal: Maximize their utilities, based on:
    - *Waiting time*: to minimize, so prefer the resource with minimal queue length
    - *Response time*: the time elapsed between the job generation and its completion
      - Depend on: the queue length, the processing capacity
    - Others, such as the accuracy of the completion time prediction
  - Used utility definition: $\rho_i = a_i T_w + (1 - a_i) T_{exc}$
    - T_w waiting time, T_{exc} job execution time
    - a_i randomly chosen for each agent so as to account for the heterogeneity

The Model:

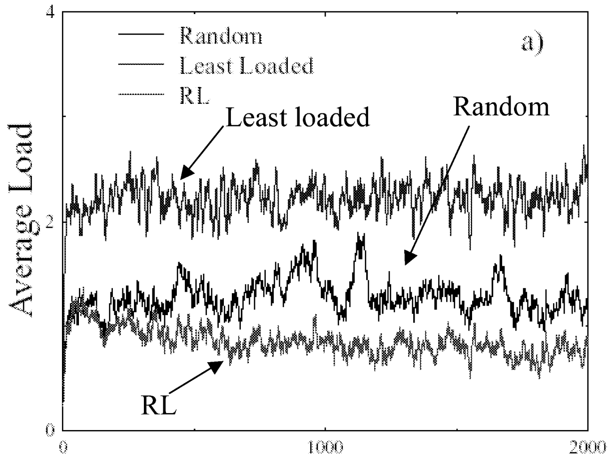
Resource Selection

- Problem: How the agents select resources?
- Solution: Using reinforcement learning, specifically, Q-learning
- Q-learning formulation:
  - Each possible action (selecting a specific resource) <==> a Q-value, indicating the efficiency of the resource in the past
  - For a new job, choose a resource with the *ε*-greedy rule (usually ε is small):
    - With (*1-ε*), choose the resource with the highest Q-value (ties are broken randomly)
    - With *ε*, randomly and uniformly choose in the other resources
  - After a job is completed, calculate its utility ρ_i, then its reword:

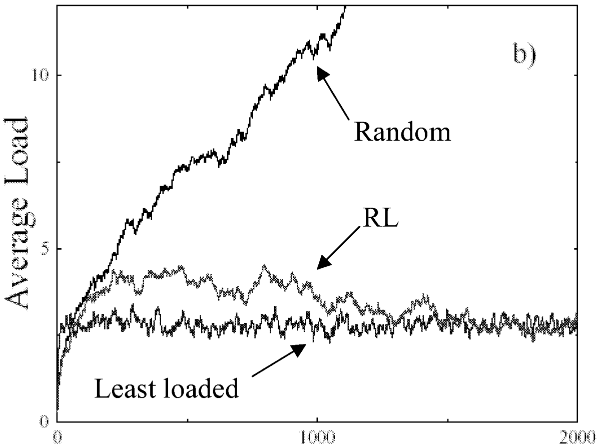
    $$r = sign(\langle \rho_i \rangle - \rho_i)$$

    where $\langle \rho_i \rangle$ is the average utility among all submitted jobs.
  - Update Q-value: $Q_{i,t+1} \leftarrow Q_{i,t} + \alpha(r - Q_{i,t})$
- Other selection rules for comparisons:
  - Random selection: choose randomly and uniformly among all resources
  - Least loaded: choose the least loaded resource to submit a job
    - Requirement: the up-to-date global knowledge about the load of all resources

Experimental Results



Job arrival rate P = 0.15

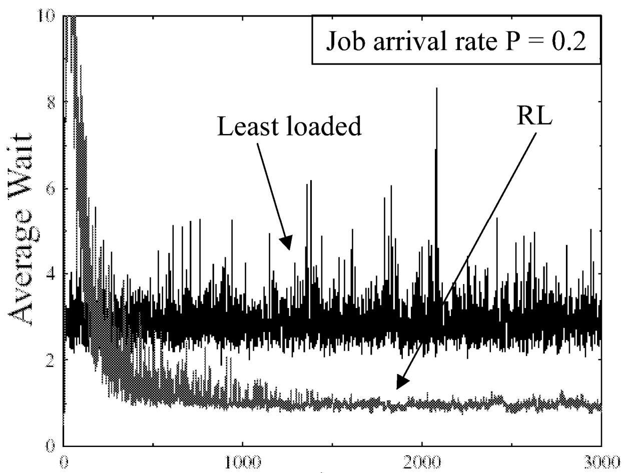


Job arrival rate P = 0.2

- Experimental setting:
  - 1000 agents and 250 resources
  - Job arrival rate P = [0.1, 0.2]
  - Job length [10, 1000]; randomly and uniformly choose
  - Resource capacity [350 650]
- Observations:
  - For small job arrival rate P (Figure a): Random selection performs better?
  - For large job arrival rate P (Figure b): Random selection is deteriorated!
  - RL is more efficient than random selection in resource allocation

Experimental results on:

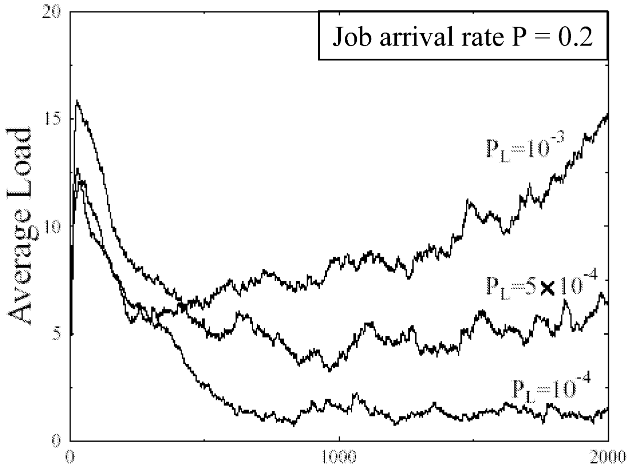
Average waiting time



- Observations:
  - RL has a learning phase where RL performs worse than least loaded, after that
  - RL performs much better than least loaded
  - RL without global knowledge vs least loaded with global knowledge

Experimental results on:

Effect of dynamic agent population



- Observation: The user might dynamically join or leave a VO.
- Problem: What's the effect of this dynamics on the RA mechanism?
- Means: 1) At each time step, an agent leaves its VO with a probability P_L.
  2) For each leaving agent, add a new agent, that has to start its learning from zero.
- Results: 1) For small P_L, the impact is negligible;
  2) For large P_L, the performance was deteriorated due to the large number of new agents.

Discussion

- RL performs better than random selection, which is commonly used, currently
- RL provides better adaptive behavior because each agent learns from its response from the environment

Future Work

- Develop some external resource discovery system so as to identify the action space of agents in RL
- Study more complex jobs, which require co-allocation of different resources

Problems of the Model

- Problem1: if there are numerous resources, the action space of an agent is very huge. The storage for Q-values become a problem.
- Solution1: each agent maintains Q-values for only two kinds of actions: 1) n actions with the highest Q-values; and 2) m newly visited actions
- Solution2: categorize resources into different types according to its properties in consideration, e.g., computing power

- Problem2: For each leaving agent, add a new agent who has to restart to learn.
- Solution1: maintain a profile for a user. For a new user, the corresponding agent needs to learn from scratch. Otherwise, he/she does not need to learn from zero.
- Solution2: provide a certain form of information sharing between agents, e.g., the agents from the same grid node can share some of their information.