Static Resource Allocation

Mark Towers

June 2019

1 Problem Statement

In prior research into resource pricing for resource allocation within cloud computing, job's considered have had fixed resource requirements that server's must fulfill. However in this research we consider job's that have requirements that must be fulfill with a deadline so that the resource allocation must allow the deadline consider to be true. We have then developed a greedy algorithm to solve the problem to maximise the social welfare where the total utility is known of ecah jobs. However in real life then server's that run the job would want to be payed to run a job so we have developed new auction.
In the new section, we will describe the problem case and how we generated a model.

2 Problem Case

2.1 Variable

There are J jobs, indexed with $j = 1, \ldots, J$ and I servers, indexed with $i = 1, \ldots, I$.

- $x_{i,j} \in \{0, 1\}$ indicates whether the job j was done on server i
- $s_{i,j}$ the rate that the program is loaded at (MB/s)
- $w_{i,j}$ the rate of computation (TFlop/s)
- $r_{i,j}$ the rate that the result's data is sent back (MB/s)

2.2 Constants

Server - i

- Maximum storage - S_i (MB)
- Maximum computation capacity - W_i (TFlop/s)

- Maximum communication bandwidth - R_i (MB/s)

Job - j

- Required storage - s_j (MB)
- Required computation capacity - w_j (TFlop)
- Required data for results - r_j (MB)
- Utility - U_j (\$)
- Deadline - D_j (s)

2.3 Optimisation

$$max \sum_{j=1}^{J} U_j x_{i,j} \qquad \forall i = 1, \ldots, I \tag{1}$$

2.4 Constraints

Job to server allocation

$$\sum_{i=1}^{I} x_{i,j} \leq 1 \qquad \forall j = 1, \ldots, J \tag{2}$$

$$x_{i,j} \in \{0, 1\} \qquad \forall i = 1, \ldots, I; j = 1, \ldots, J \tag{3}$$

Server resource available

$$\sum_{j=1}^{J} s_j x_{i,j} \leq S_i \qquad \forall i = 1, \ldots, I \tag{4}$$

$$\sum_{j=1}^{J} w_{i,j} x_{i,j} \leq W_i \qquad \forall i = 1, \ldots, I \tag{5}$$

$$\sum_{j=1}^{J} (r_{i,j} + s_{i,j}) x_{i,j} \leq R_i \qquad \forall i = 1, \ldots, I \tag{6}$$

Process completed within deadline

$$\frac{S_j}{s_{i,j}} + \frac{W_j}{w_{i,j}} + \frac{R_j}{r_{ij}} \leq D_j \qquad \forall i = 1, \ldots, I; j = 1, \ldots, J \tag{7}$$

Resource usage

$$0 \leq s_{i,j} \qquad \forall i = 1, \ldots, I; j = 1, \ldots, J \tag{8}$$

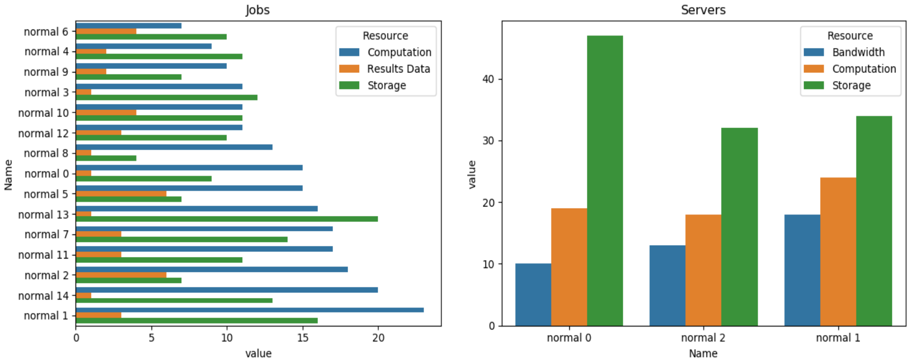
$$0 \leq w_{i,j} \qquad \forall i = 1, \ldots, I; j = 1, \ldots, J \tag{9}$$

$$0 \leq r_{i,j} \qquad \forall i = 1, \ldots, I; j = 1, \ldots, J \tag{10}$$

2.5 Problem Case Explanation

- Equation 1 is the objective function that maximises the sum of the job utility for jobs completed.
- Equation 2 and 3 enforce that a job is only done on a single server.
- Equations, 4 to 6, ensures that the server resource used are within the maximum resources available.
- Equation 7 enforces that the job will be completed within the deadline on only the servers that is a job runs on.
- Equations, 8 to 10, ensures that resource speeds are within a valid range of greater than 0

2.6 Example cases



(a) Example jobs

(b) Example Servers

Figure 1: Example jobs and servers

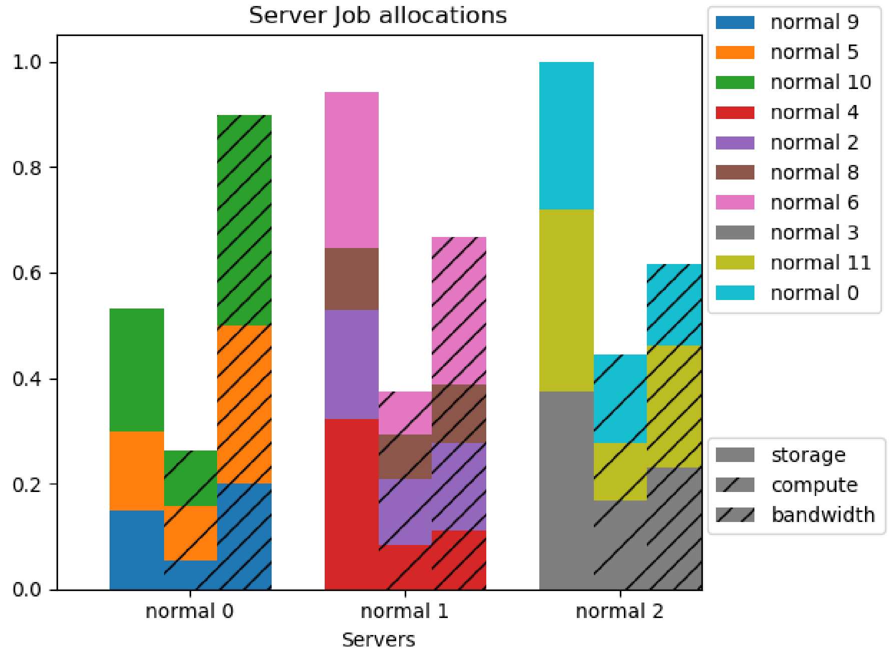
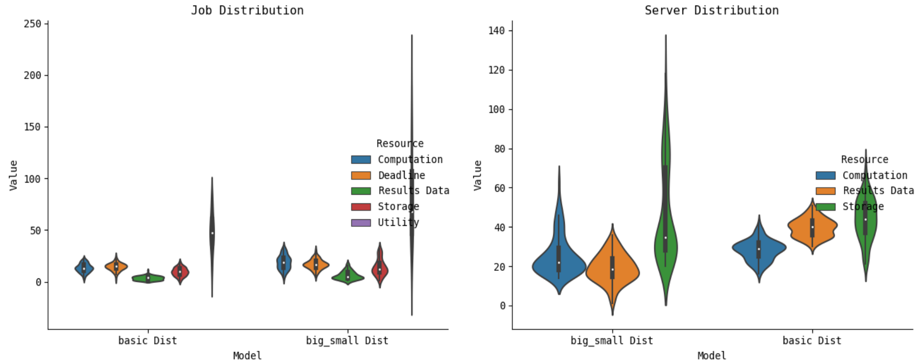


Figure 2: Example allocation

2.7 Model creation

To create each of the jobs and servers we choose a mean and standard deviation for each attribute that is then used to generate a random number from a normal distribution. We normalise the value generated to make it an integer and the max of 1 and the value to make sure it is greater than zero.

Job Attribute	Mean	Std	Server Attribute	Mean	Std
Storage	10	4	Storage	45	10
Computation	15	5	Computation	30	5
Results data	5	4	Bandwidth	40	5
Utility	50	20			
Deadline	15	4			



(a) Example job distribution

(b) Example Server distribution

Figure 3: Example jobs and servers distribution

3 Greedy Algorithm

To solve a problem case is NP-Complete as it is a multiple multiple-dimensional 0-1 knapsacking problem preventing fully polynomial time approximation. So we have developed a greedy algorithm that creates a near optimal solution.

To do this requires three functions: value density, server selection policy and resource allocation policy. The value density function is used to calculate how important a job is based on it's attributes that can used to order the jobs. Once the jobs are ordered on their values, then a server is selected for which server to choice to run the job on. As the deadline constraint exists then we must also calculate how much resources should be allocated to the job by the server.

3.1 Greedy implementation

```
job_values = sorted(((job, value_density.evaluate(job)) for job
    ↪ in jobs), key=lambda jv: jv[1], reverse=True)

# Loop through all of the job in order of values
for job, _ in job_values:
    # Allocate the server using the allocation policy function
    allocated_server = server_selection_policy.select(job, servers
        ↪ )

    # If an optimal server is found then calculate the resource
        ↪ allocation policy
    if allocated_server:
        value, (s, w, r) = resource_allocation_policy.allocate(job
            ↪ , allocated_server)
        job.allocate(s, w, r, allocated_server)
        allocated_server.allocate_job(s, w, r, job)
```

4 Greedy Example

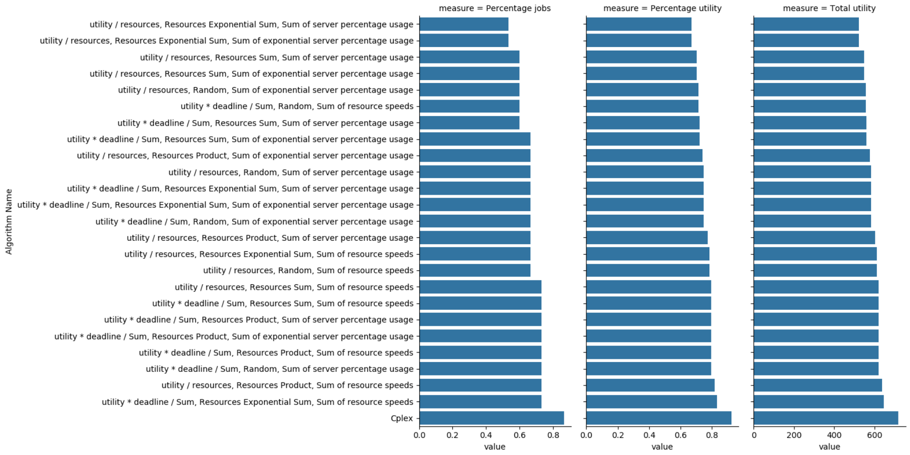


Figure 4: Algorithm Results

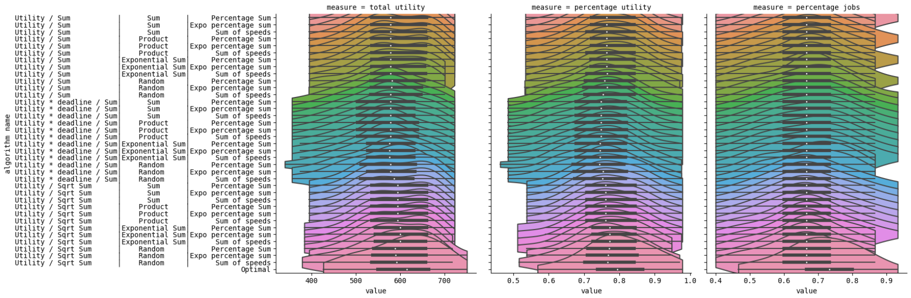
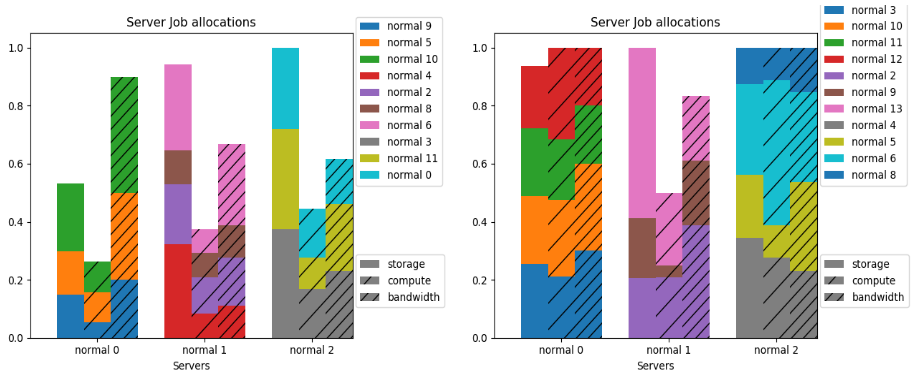


Figure 5: Repeat Algorithm Results



(a) Example Greedy Allocation

(b) Example Optimal Allocation

Figure 6: Example jobs and servers

5 Auctions

The main idea was that currently most job work out how much resources that they require to run and so bid for that amount of required results however in doing this they are unaware of other jobs and their requirements. Therefore there is both a competition and cooperation involved we believe that can increase the total social welfare for everyone if people work together while in the mean time allowing the job to not give away its utility as in VCG auctions. In real life then server will want to payed to the work that it does however to do this is not easy as the server allocate the resource price dynamically and as a job may not want to reveal its utility. We have created an iterative VCG auction.

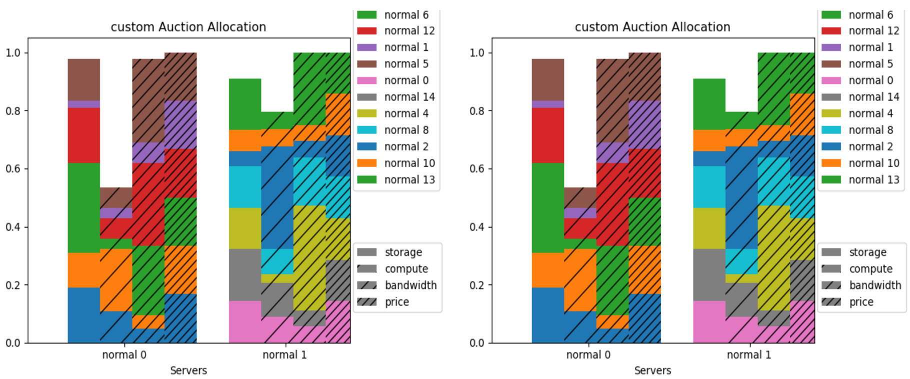
5.1 Auction algorithm

```
unallocated_jobs = jobs
while len(unallocated_jobs):
    job = choice(unallocated_jobs)

    # Calculate the minimum job price on all of the servers
    job_price, allocation_info = min((evaluate_job_price(job,
        ↪ server, epsilon=epsilon)
                                     for server in servers), key=
                                         ↪ lambda bid: bid[0])

        if job_price <= job.utility:
            # Uses the allocation info to create the new
                ↪ allocation on the selected server
            allocate_job(job_price, job, allocation_info,
                ↪ unallocated_jobs)
        else:
            # Remove job as there are minimum server is greater than the job's utility
            unallocated_jobs.remove(job)
```

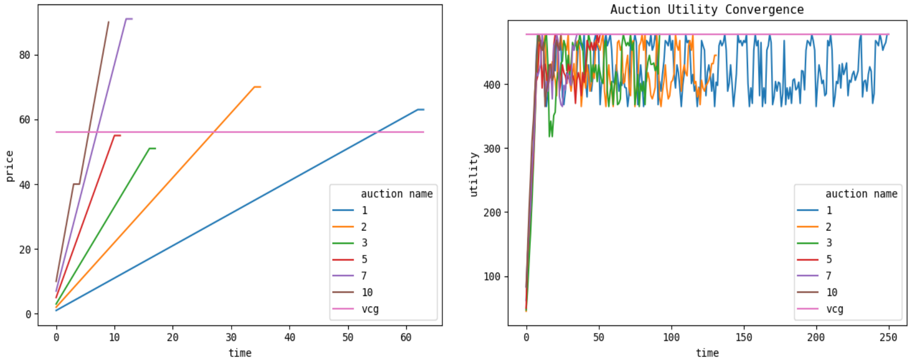
6 Auction Results



(a) Example VCG Allocation

(b) Example Custom Allocation

Figure 7: Example of auction results



(a) Auction Price Convergence

(b) Auction Utility Convergence

Figure 8: Auction Convergence