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 job's owner would then request from a cloud service. However this could result in different job's fighting over a single resource instead of sharing the resource fairly. In this research we consider job's that have requirements that must be fulfill with a deadline allowing the cloud services the ability to dynamically allocate resources depended how the usage of it's resources from other jobs and share resources more effectively. We have then developed a greedy algorithm to solve the problem to maximise the social welfare where the utility is known of each jobs. However in real life, server's that run a job would want to be payed to run a job so we have developed an iterative auction.

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

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

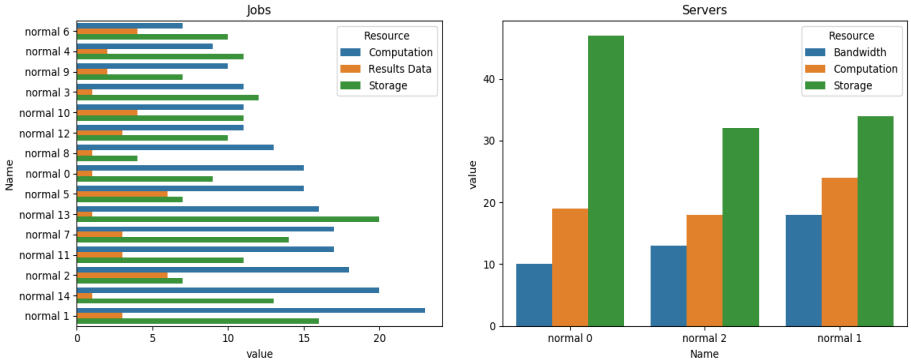
$$0 \leq w_{i,j} \qquad \forall i = 1, \ldots, I; j = 1, \ldots, J \tag{9}$$

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

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

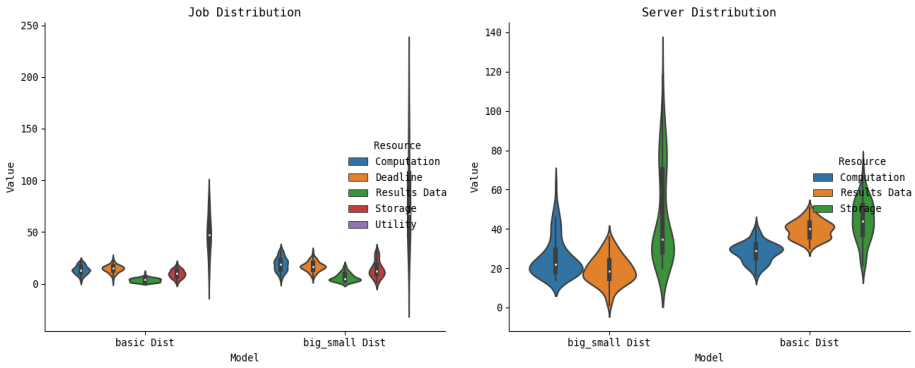
(b) Example Servers

Figure 1: Example jobs and servers

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 check that the value is greater than zero.

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



(a) Example job distribution

(b) Example Server distribution

Figure 2: Example jobs and servers distribution

3 Greedy Algorithm

The problem case layed out below is a multi multi-dimensional 0-1 knapsacking variation which has been shown to be NP-Hard to solve. Therefore while a solution can be found through integer programming, it is extremely slow to do and become impossible to solve for problem cases with more than 15 jobs and 3 servers on a regular computers.
This means that to calculate the optimal solution, with full knowledge, then a greedy algorithm must be developed. To do this, I have created two greedy algorithm that are compared and explained below.

3.1 Greedy implementation

The first algorithm is a three stage process where the jobs are sorted on how good they are based on a metric like the ratio of utility to resource's required. For each jobs, a server is selected for the jobs to run on based on the another metric like which server has the least resources available. Resources are then allocated to a job based on a third value policy like the percentage of the available resource that the allocated resources would use.

This allows for a large amount of experimentation due to the number of permutations that exists for different policies of the different stages however any errors are propagated through the system more easily and the job value doesnt change has the server's resources are allocated off.
Below is the algorithm used to generate the solution written in python with a large number of policies available to try.

```
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)
```

3.2 Matrix Greedy implementation

The second algorithm is significantly simpler as it only uses a single policy and hopes to prevent many of the possible problems found with the first algorithm. I believe that it will be easier to prove any approximations of the algorithm with this version due to having a single policy compared to three of the last version. This algorithm works by thinking about the resource allocation first instead of last and so for each job and server the best resource allocation is found using a metric like the product of the job utility and the percentage of the server available resources after allocation. This is then used to generate a matrix of jobs and servers allocation with the value being the value of the best resource allocation for the job and server. From this matrix then the max value is found of this with the job then being allocated to the server and the job removed from the matrix. This is iteratively done updating each time till all of the jobs are allocated or none of the jobs can be allocated.
Due to this simplification with only a single policy, I believe that this should make the algorithm better however as the policy must include the job utility and some function of the server available resources and the resource speed this is less simple to do. Therefore this is something I am still experimenting with and to prove if any ϵ approximations are possible.

```
def allocate_resources(job: Job, server: Server, value_policy:
    ↪ MatrixPolicy):
    return max(((value_policy.evaluate(job, server, s, w, r), s, w
        ↪ , r)
                for s in range(1, server.available_bandwidth + 1)
                for w in range(1, server.available_computation + 1)
                for r in range(1, server.available_bandwidth - s +
                    ↪ 1)
                if job.required_storage * w * r + s * job.
                    ↪ required_computation * r +
                s * w * job.required_results_data <= job.deadline *
                    ↪  s * w * r), key=lambda x: x[0])


def matrix_greedy(jobs: List[Job], servers: List[Server],
    ↪ value_policy: MatrixPolicy):
    unallocated_jobs = jobs
    while unallocated_jobs:
        value_matrix = []
        for job in unallocated_jobs:
            for server in servers:
                if server.can_run(job):
                    value, s, w, r = allocate_resources(job, server
                        ↪ , value_policy)
                    value_matrix.append((value, job, server, s, w,
                        ↪ r))

        if value_matrix:
            value, job, server, s, w, r = max(value_matrix, key=
                ↪ lambda x: x[0])
            job.allocate(s, w, r, server)
            server.allocate_job(job)
            unallocated_jobs.remove(job)
        else:
            break
```

3.3 Greedy algorithms results

Results

4 Auctions

While the greedy algorithm allows us to find near optimals for problems where we have perfect information about the problem case like the job utility, this

may be private information that the job owner doesnt want to reveal also server would want to get paid for that work that they do.
We propose a iterative auction where the job owner doesnt have to reveal it's true utility. In normal VCG auction, then the auctioneer will solve the problem to maximise the social welfare (total utility of allocated jobs) and must find the optimal solution. Then for each job and server, the job or server is removed from the problem case and the optimal is solved for that as well with the server's revenue it is paid and the job's cost being the social welfare of the optimal solution minus the social welfare of the optimal without the job or server. This is extremely slow and unscalable because of this so is not used often however has the properties that it is individually rational, truthful bidding and incentive compatible.
Our iterative auction uses the idea of VCG so that when a job asks to run on a server then the server will calculate current revenue of the jobs running minus the revenue if the new job must be running on the server plus a price increase factor. This is then the price for the job to run on the server as the new allocation with the job would be to a greater revenue for the server than is currently allocated.

4.1 Iterative Auction algorithm

```
unallocated_jobs = jobs
while len(unallocated_jobs):
    # Select a job, can be at random
    job = unallocated_jobs[0]

    # 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])
        # Check if the job can pay the minimum price
        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 the job can be run ever at a price
                ↪ lower than the job's true utility
            unallocated_jobs.remove(job)
```

5 Iterative Auction Results

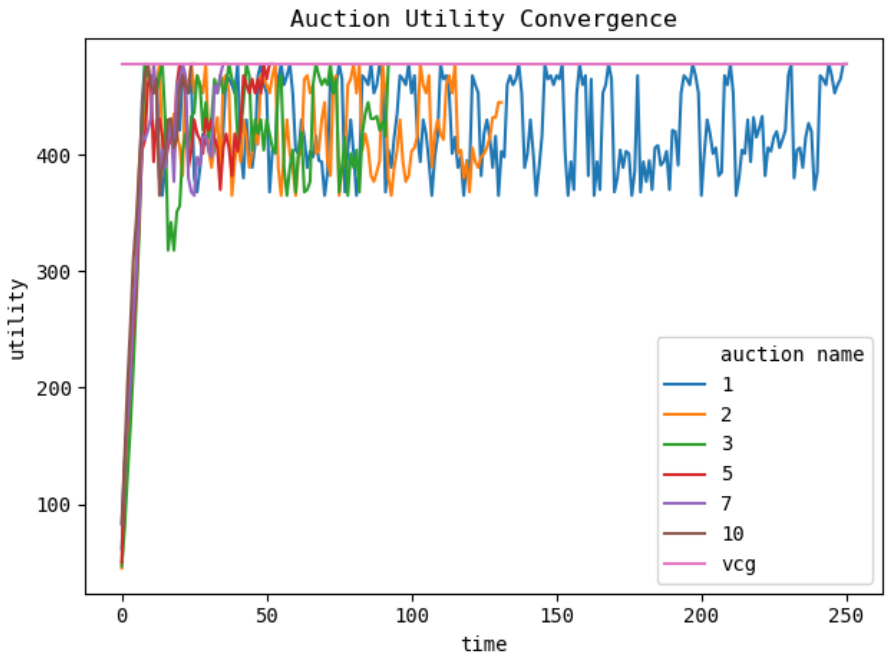


Figure 3: Auction Utility Convergence