A Data Center Network Allocation Scheme

Sarah Gontarek, Ami Greene, Kate Rudolph

1 Introduction

As available data grows exponentially, the need to run large computational jobs is growing as well. This has led to a new market for renting space in data centers. Small companies can rent space in the data centers of larger organizations such as Amazon or Rackspace, enabling them to run large jobs without maintaining their own datacenter. Users pay rent based on the number of machines they use and the duration of the job.

As running these outsourced jobs becomes increasingly common, optimizing the time and money spent on them becomes an important consideration. In most cases, the speed of these jobs is limited by the speed of the networks as machines send large amounts of data to each other much more slowly than the data is generated. To optimize the speed and cost of these jobs, the user must place their machines in the datacenter to maximize the network throughput between machines.

This paper describes a simple but effective algorithm that can be used to optimize the speed, and thus the cost, of a job run in the data center of DP2.[1]

2 Overview

2.1 Assumptions

Our first and **most important** assumption is about the pricing in the data center: we assume that we can be charged for fractional minutes of time on a given physical machine. We realize this assumption has a huge effect on the nature of the problem we are asked to solve, but we state[2] this assumption clearly at the outset because it clarifies the problem domain.

---

[1] http://web.mit.edu/6.033/www/assignments/dp2.html

[2] That is, we are able to change the physical machine that a VM is situated on multiple times per minute, and pay the data center for the total number of uptime minutes of that VM. If this were not true, our design would fail horribly. However, it's reasonable to assume we can move VMs around frequently because the network traffic and user population may change in a single second, and if our design cannot respond to those changes there is little point measuring the network traffic to the level of fidelity that the specification suggests. On Thursday morning, we received information on Piazza informing us that a ceiling function would be imposed on VM operation such that if any VM occupied any physical machine for less than a full minute, the user would be charge for at full price for an entire minute. This would imply that migrating a single VM 4 times in a single minute would lead to a charge of four minutes. This substantially changes the design problem. Given how late this information was revealed, and what seemed to be some amount of disagreement or uncertainty on the part of the TAs, we chose to proceed with our design solution in its original form, which was informed by the written design problem and does not account for this feature.

Additional assumptions:

- Our application will need 6 to 600 VMs. (Reasonable because total capacity is only 4000 VMs)
- Average network latency is 1ms, worst case is 10ms when queues are long (from assignment, TA office hours)
- place and random_place have latency about 10ms, with the worst case being 100ms
- There are at least N+10 spaces available on the data center
- Machines are numbered sequentially; i.e. machines 1-48 belong to group 1, 49-96 to group 2, etc.
- VMs can address each other by VM number: we need not keep track of the physical location of each VM

2.2 Strategy

To implement our measurement and placement algorithms, we will allocate one to five additional VMs beyond the N application VMs. One of these additional VMs, called the *master* VM, will consolidate measurements and coordinate placement of the application VMs. There may also be some number of *scout* VMs, depending on N. The exact number of scouts is explained and justified in the Section 3.1.1. The number of additional VMs will be a small percentage of the total, and increased speed will more than compensate for the additional cost.

The master will use the `machine_occupancy` function on every physical machine in the data center to determine where space is available. Then it will send scouts to sample the throughput between different groups in the data center. The details of this measurement are presented in Section 3.1.

The key idea of the algorithm is that the entire computation can only finish as fast as the slowest transfer, and thus we should maximize the throughput between the VMs with the most bytes left to transfer. Each second, the entire system will run a reassignment algorithm, coordinated by the master. The master will determine how many bytes remain to be transferred between each pair of VMs. It will sort this list of pairs from most bytes remaining to least, placing the machines in groups where there is most space. Once these spaces are filled with VM groupings, it will continue to place VMs in remote spaces with the least network congestion. The details of this placement algorithm, and a justification for the one-second timing, are presented in Section 4.

This design does significantly better than a random placement, and the greedy approach is significantly simpler than any algorithm which attempts to measure all possible throughputs and perfectly optimize the machine placement. A comparison to these and other alternative designs is included in Section 5.

3 Measurement

In order to optimize for space on the data center and network conditions, we need a way to measure both machine vacancies and throughput between different locations. The two different measurement schemes we used are explained below: to measure space, the master will simply call the `machine_occupancy` method on every machine in the data center; to estimate

network congestion, we use scout VMs to randomly probe inter-group links in the data center network.

3.1 Network Measurement

The master sends out all the scout VMs to randomly chosen groups, such that no two scouts go to the same group. The probability of a scout being placed on a group is weighted by the amount of free space on that group. The placement of the scouts is explained in Section 4.4.

Once placed, each scout will use the `tcp_throughput` function to sample the throughput to each of the other scouts, averaged over a window of 100ms. The scouts report the these observations back to the master. The messages they send to measure throughput are small and thus do not significantly contribute to network congestion for the data transfers required by the application. After 100ms, as soon as the scouts have completed their measurements, the master will randomly assign them to new groups. Our assumptions indicate that the worst case latency for for moving scout VMs is 10ms, and that expected latency is closer to 1ms.

We can assume network traffic and machine occupancy don't change appreciably on a subsecond time scale. Thus, any measurement made within the past second is still fresh -- all older measurements are stale. This means that the master has time to re-randomize and re-assign the scouts about 10 times before it must make placement decisions based on the new information. After one second - or ten scouting assignments - the information becomes old and is discarded.

The master stores all fresh network throughput measurements in a network table. Each line in the table contains an ordered pair of groups and the throughput values observed from the first group to the second group. Once the information becomes stale, the master deletes it from the network table.

3.1.1 Number of scout VMs

The number of scout VMs will be determined according to the following table:

# Application VMs	# Scouts	% Overhead
0-24	0	4.2% or more
25-95	2	12%-3.2%
96-191	3	4.2%-2.1%
192+	4	2.6% and lower

Table 1. The % Overhead column is the percentage of additional VMs beyond the application VMs that our scheme will use, including the single master and all scouts. It is computed as

$$\frac{1 + \#Scouts}{\#Application\ VMs} \times 100\%$$

We have carefully considered these values. When 24 or fewer application VMs, 1/8th the maximum capacity of a single network group, with only moderate crowding it's highly probable that all VMs could be placed on the same group by the placement algorithm. Even if all

application VMs cannot be placed together, the relative overhead cost of scouts is enough that the potential benefit does not compensate. This is discussed further in Section 5.4.

There is no level at which we purchase a single scout VM. The scouts are only measuring throughput among one another, so a lone scout cannot make any measurement. One could imagine a scheme in which a single scout measures throughput back to application VMs. However, the traffic introduced by these measurements would slow down the application VMs, and we wouldn't be able to test traffic between groups on which we have no VMs. So two VMs are a minimum.

Beyond two scouts, the levels are chosen such that the percentage overhead of all additional VMs is approximately 2-4%. As explained in Section 5.4, the additional information provided by the scouts is enough to save much more than 2-4% of the total computation time, so the additional investment pays off in shorter runtime.

We never place more than 4 scouts. Since all scouts measure throughput to all other scouts, measuring throughput to many other scouts simultaneously would result in enough traffic to create an artificial bottleneck on outgoing links. This would make our measurements inaccurate, so we cap the number of scouts at 4.

We could use the master as a scout, moving it around every 100ms and measuring the throughput between it and all other scouts. However, we reject this idea because the throughput measurements would congest communication for the master. The master's communication to the rest of the network needs to be robust in order for it to carry out the space measurement and the placement algorithm, and excess congestion could slow down our entire algorithm. Therefore, the scouts are separate from the master.

3.2 Space Measurement

While the scouts measure network conditions, the master will be making measurements of all available space in the data center. Where the network measurement scheme is complex, this space measurement is very simple. Every second, the master iterates through every physical machine m on every group in the data center and calls `machine_occupancy(m)`, storing the results.

The actual process of measurement will easily complete within 1 second as there are only 1152 physical machines in the data center, and `machine_occupancy` is a very fast function, upper bounded by the latency of network communication, which is on average about 1ms. Since all requests can be sent over the network in parallel, the data can be received at the master well within the 1-second time limit for making decisions about placement.

3.3 Progress Measurement

The master will also track of the progress of each pair of machines by calling `int progress(u,v)`. These measurements are also inexpensive. The master subtracts the progress between every pair of nodes from the transfer requirements in the initial matrix to determine the number of remaining bytes for each pair. It constructs a list of all N^2 pairs of VMs, sorted from most to least remaining bytes, to be used in the placement algorithm.

3.4 Measurement Overhead

When making measurements of the data center network throughput, we must make sure we don't generate too much traffic and slow the network down. This is why we use a randomized measurement scheme: to get a good sampling of the picture of network congestion without the cost of an exhaustive or comprehensive measurement. This allows measurement overhead to adapt to the size of our job. Our measurements use only the `tcp_throughput` method provided, and only send small messages back to the master to communicate this information.

The space measurement requires the master to send 1152 machine_occupancy requests and receive 1152 responses. Since the response is only a number between 0 and 4, it easily fits in one packet. These approximately 2000 packets are sent out and received from every single machine in the data center, widely distributing the traffic over all possible paths through the data center. The link to the master's own group may grow congested, but since a packet can be as small as about 100 bytes, resulting in only 200kB of total messages, this is not enough to significantly hamper a 10GB/s link.

The movement of the scouts does not create congestion, because the actual data transfer needed to move a VM is negligible compared to the data transfers the VMs are doing amongst each other.

The only potentially problematic aspect of the network measurement is the traffic generated by the scouts between each other. The data they send for measurement purposes could mildly contribute to network congestion. Every 100ms, there are S^2 such measurements running, where S is the number of scouts, as determined using Table 1.

However, if the data the scouts send is enough to contribute to network congestion for our actual application VMs, then it means the throughput is already very low along that path. Thus, it was a good thing to have measured it because in the next placement, we'll be able to avoid that bad path. Any increased network congestion caused by our measurement is therefore justified, because it helps us avoid disadvantageous network paths in the future.

4 Placement

This section discusses how the master VM uses measurements described in Section 3 to place VMs within the data center. Our approach to this algorithm is guided by a few key principles.

1. Our VMs should be close together, to take advantage of the fast intra-group connections. If they cannot all fit in a group, the best alternative is to place VMs in the group where they will have the best throughput to the VMs with which they most need to communicate.

2. Pairs of VMs with the most data left to be transmitted should share good connections.

3. Available space on a group is often an indicator of intra-group throughout.

4. Space measurements are cheap, but throughput measurements are more expensive. Therefore throughput between all pairs of groups should not be measured all the time. It is

more important to measure the throughput of groups with more free space, since they are more likely to have desirable throughput.

5. Since the data center will be changing every second, and there is a very low cost to moving VMs, the VMs should be moved every second to take advantage of new data center conditions.

Our algorithm uses the master node described in Section 2.2 to freshly place all application VMs once every second. In addition, the master also places the scouts up to ten times per second, using a simpler algorithm that depends on the occupancy of groups. This algorithm is described in Section 4.4..

4.1 Data Structures

The master VM uses a few simple structures to store the measured data. The number of VMs available in each group is stored in a dictionary called `group_space`, with key, value pairs of (group number, available space). `group_space` counts in its values the space occupied by the user's VMs, so it does not get decremented as VMs are placed. However, full groups are removed from the list (see `place_group(vm, g)` in the next section). The throughput information is stored in a table `throughput_table` ordered from high throughput to low throughput. An additional dictionary, `vm_places` is maintained which stores (VM number, machine number) pairs. Furthermore, the master has a list `data_pairs` of (VM a, VM b, `data_remaining`) sorted by data remaining from high to low.

4.2 Helper Functions

Several helper functions are used in our placement algorithm. They are relatively simple, but the edge case behavior is important, so they are described here.

- `machine_id get_machine(vm):` returns the machine of the specified VM using `vm_places`

- `group_id get_group(vm):` returns the group number of the specified VM using `vm_places` and the known structure of machine numbers within the data center (see Section 2.1)

- `(machine_id, IP_addr) place_group(vm, g):` Similar to the provided `place_random(vm)`, except it places a VM randomly within the given group *g*. Iterates through machines within *g* trying to place VM. Returns the machine number where the VM was successfully placed, or null if no placement was possible. If placement was not possible, group *g* is removed from the `group_space` dictionary. This function will be no slower than the provided `random_place`, but provides more control.

- `group_id find_highest_throughput(g):` Returns the ID of the non-full group with the highest known throughput to group *g* using `group_space` and `throughput_table`. If multiple groups have the same throughput to *g*, returns the one with the most fre space.

If free space fails to differentiate the groups, one is selected randomly. If there are no entries in the table for g, returns the result of `most_space()`

- `group_id find_most_space()`: Returns the id of the group with the most free space that also exists in `throughput_table`, using `group_space`. If two groups have the same amount of space, returns the one that shows up first in `throughput_table` (i.e., the one with the highest throughput to another group). If throughput fails to differentiate groups, one is chosen randomly.

4.3 Initial Placement

Initially, all of our VMs, including the master, are loaded into the data center machines using `random_place`. The master VM's location is never changed.

4.4 Scout Placement

The master uses scouts to gather space measurements (see Section 3.2) and thus must place these scout machines in different groups to gather data. Measuring the throughput between each pair of groups in every second would consume too many resources, so scout VMs are randomly placed throughout the data center, with a bias toward groups that show low occupancy.

The probability distribution goes as the square root of free space. (A distribution that was linear with free space would revisit the same open groups too frequently, while a distribution that went with the log of free space would not show enough preference for open groups.)

$$p(Placement = GroupX) = \frac{\sqrt{(1 - machine\_occupancy_X)}}{\sum_i \sqrt{(1 - machine\ occupancy_i)}}$$

The pseudocode for the scout placement algorithm is shown in Figure 1

```
unscoutedGroups[]
def p(group g):
      num = sqrt(1-group_space[g])
      denom = [sqrt(1-group_space[i] for i in unscoutedGroups]
      return num/denom

Every 100 ms:
      // use this list to ensure that two scouts don't get placed in same group
      unscoutedGroups = [1, 2, 3, 4, 5, 6, 7, 8]
      for each scout:
            select group g based on p(g)
            place_group(scout_id, g)
            remove g from unscoutedGroups
```

Figure 1: the psuedocode for the scout placement algorithm

4.5 Application VM Placement

We have developed a greedy placement algorithm for the master that can be executed quickly and achieves reasonably good results. Every second, the master runs this placement algorithm. The algorithm iterates through the sorted pairs of VMs (described in Section 3.3). For each pair, if neither are placed, the algorithm places them in the group with the most 'space', where space consists of machine vacancies plus space already occupied by our own VMs. If one is already placed, the algorithm tries to place the second as 'close' to the first as possible, where closeness is in terms of physical placement (same machine, same group) and throughput (to the group where the first VM is located). This algorithm is depicted in much greater detail below.

```
Figure 2 (pseudocode):
      vm_places = {}
      for (a, b, data_remaining) in data_pairs:   // a and b are VMs
            // if both are placed, do nothing
            if (a is in vm_places) and (b is in vm_places):
                  pass

            // if just a is placed, put b close to a
            elif a is in vm_places:
                  place_near(a, b):

            // if just b is places, but a close to b
            elif b is in vm_places:
                  place_near(b, a):

            // if neither of them are placed, put them in the group with
            // the most space (space = free space + user VMs)
            else:
                  // try to put both in the group with most space
                  // repeat until can put them in a group together
                  while (len(group_space) > 0) and (not tryPlaceA):
                        bestGroup = find_most_space()
                        tryPlaceA = place_group(a, bestGroup)

                        // if place a, try to place b in same group
                        if tryPlaceA != null:
                              vm_places[a] = tryPlaceA[0]
                              tryPlaceB = place_group(b, bestGroup)
```

```
                              // if can place b in same group, done
                              if tryPlaceB != null:
                                    vm_places[b] = tryPlaceB[0]
                                    return
                              else:
                                    break
                        // if cannot place b in same group, place in group with
                        // high throughput to bestGroup.
                        // ideally, would be placed in the same group. But since
                        // placement only lasts a second and pairs will be
                        // reprioritized by data left, maintain simplicity
                        // over correctness
                              place_near(a,b)

// definition of helper function place_near
// trying to put VM b close to VM a
def place_near(a, b):
      // try putting b on the same machine as a
      m = get_machine(a)
      tryPlace = place(b,m)
      if tryPlace != null:
            vm_places[b] = m
            return

      // try putting b in the same group as a
      g = get_group(a)
      if g is in group_space:
            tryPlace = place_group(b,g)
            if tryPlace != null:
                  vm_places[b] = tryPlace[0]
                  return

      // try putting b in the group with the highest known
      // throughput to a's group. If fails once, try again
      while (len(group_space) > 0) and (not tryPlace):
            bestGroup = find_highest_throughput(g)
            tryPlace = place_group(b, g)
            if tryPlace != null:
                  vm_places[b] = tryPlace[0]
                  return

      // if all else fails
```

```
tryPlace = place_random(b)
vm_places[b] = tryPlace[0]
```

Figure 2: the pseudocode for the application VM placement algorithm

5 Analysis

We analyze this design in the three given use cases, and further consider a few worst-case scenarios for our algorithm. We address the fault tolerance of our system, and finally describe the cost and speed the user will experience using our placement algorithm.

5.1 Common Use Cases

This section considers our system's behaviour on the three given use cases.

5.1.1

N=100, each machine already hosts 3 VMs, data rate and transfer requirements are uniform.

In this scenario our algorithm uses three scouts and one master, and performs very well. Initially, when the algorithm generates a list of the pairs of user VMs sorted by data to transfer, the list will effectively be random. Since all groups have the same amount of space, our algorithm will pick a group effectively at random in which to begin placing VMs using `get_most_space()`. Our algorithm will them place these machines into that group, pair - by - pair. After the first group is full, the 49th VM will be placed randomly in another group using `get_most_throughput()` or `get_most_space()`, which will also effectively be random. The 50th - 98th VMs will be placed on groups chosen chosen by `get_most_throughput()` and `get_most_space()`. The first will cluster the placed VMs, while the second will spread them out randomly through the remaining groups. We will end up with one full group, and smaller clusters distributed amongst the other groups.

After the first second, our algorithm will run again. Those VM pairs that were in the same group will have completed a larger fraction of the data transfer, so this time VM pairs that were split across the groups will be placed first, in the same group, so they will communicate more quickly. Each following second, our algorithm will keep shuffling the VMs between the one main group and smaller clusters so that those VM pairs that have the most to communicate will be together in the large group.

5.1.2

N=10. All 10 VMs are in groups 1-6, very congested. Large space with high throughput becomes available in groups 19-24

In this situation, our algorithm performs particularly well. It will quickly respond to take advantage of the new space, since the master VM is constantly measuring the space available in the different groups. These space measurements are completed in less than a second, and in the next second the algorithm will place all of the VMs into the newly-available group with the most space.

With this few VMs, our algorithm will not actually use scouts, but the master will still find the large available space without the scouts.

5.1.3

All users use our scheme

In this scenario, all users will see when a new space opens up and try to place their machines there. If the users are all synchronously trying to call their placement algorithm, this behavior will be problematic. Each second, every user will get some fraction of the desirable new space. As users move out of their old groups into the new groups, more free space will become available, and the users will keep fighting for it every second.

However, users will likely be slightly out of phase with each other. In this case, the data center behavior is much more reasonable. When new space opens up, one of the users will get most of it first. Subsequent users, unable to place their VMs in the new large space, will place in the next-most-spacious group. Since users count the old locations of their machines as free space, but not the old locations of other machines, the users that do not win the new free space will tend to stay in their old spaces. Over time, users will gather their VMs into tight clusters.

5.2 Worst-Case Scenarios

As with all greedy algorithms, some cases cause our system to perform poorly. In general, our algorithm does not perform well when space does not correspond to throughput. These scenarios are unlikely but illustrate the worst case of our algorithm.

5.2.1

N<25. Most groups are partially full with VMs generating low network traffic. One group has very few VMs, but is generating lots of traffic.

Here, our algorithm will not perform optimally. We would want the user's VMs to be placed on one of the machines with less space, but less traffic. However, our algorithm will place them on the group with a large amount of space and a large amount of traffic.

The performance is not too poor though: the throughput within a group will still be high enough that the job will be completed quickly.

5.2.2

Group X has some free space and a lot of traffic, while Group Y has slightly less space and significantly less traffic. The user needs to place more VMs than will fit on a single group.

Our algorithm will place VMs on the group X, and the VMs that do not fit in group X will have very low throughput. However, the VMs in the group together will have a high throughput, and in the next second the VMs that had poor throughput will be shuffled into the group. Here our algorithm does not provide optimal performance, but it still provides an acceptable solution.

5.3 Fault Tolerance

Our system may appear vulnerable to failures of the master. However, we are guaranteed machine reliability and availability by the data center. At worst, the network

congestion to the master might significantly slow its message-passing capabilities. However, since the messages the master must send and receive are all small, they would get through even in a highly-congested situation. It is not a problem if any other VM experiences extremely high congestion in one round of placement, since they will be moving in the next second and our algorithm will ensure they move somewhere with lower congestion if lower congestion exists. If a scout VM's information does not make it back to the master, there will simply be one fewer row in the network table, which is not problematic as long as it occurs infrequently.

5.4 User Experience

We have tailored our design to minimize overall economic cost to the user. Based on our assumption that the data center will only accept complete jobs for which it has room, allowing all VMs start running at the same time, we tried to minimize cost by reducing total runtime and minimizing the number of extra VMs used.

Given that all of our purchased VMs must continue to run until the last transaction is complete, our algorithm prioritizes communication between pairs of VMs with the most remaining bytes to transmit. This ensures that there is never any individual VM with a lot of information left to transmit once all other jobs are complete. This can be summarized as a 'weakest link' approach, where our algorithm prioritizes uniformity of work left, so that there is no weak link to slow down the entire job.

The percentage of extra VMs is around 2-4%, and never more than 12% in the worst case. (See the table in Section 3.1.1) However, the presence of these coordinating VMs gives us far more than a 2-4% savings on the total time we pay for in the data center, more than justifying the additional cost.

The master VM is necessary to ensure that application VMs are efficiently grouped, regardless of the total number of application VMs that are used. Even in cases where parallel communication is restricted to a small number of application VMs, where the marginal cost of including computational overhead is highest, a master can shorten the total run time (and total savings) by several orders of magnitude. As an example, say our job requires 6 VMs. Initially, with no managerial overhead, all VMs are randomly distributed among 24 groups and most VMs probably do not share a group with another VM in the job. This means that unless they are coordinated into a cluster, the throughput of the slowest transfer will certainly be less than 10 Gb/s (which is the capacity of inter-group links). However, if a master is able to effectively cluster them onto the same machines in the same group and rearrange them every second, then the average throughput experienced by the slowest transfer should be significantly more than 100 Gb/s, which equates to a more than 10-fold reduction in net cost. This more than outweighs the 17% marginal cost of including this VM in the design.

The scout VMs have a lower ROI than the master, but still improve performance well beyond their marginal cost. In jobs that use 25 or more VMs, where there is a non-trivial chance that clustering in a single group is not possible, the 10*S*(S-1) observations per second made by the S scouts produce valuable insight about alternative options. This allows us to ensure with fixed confidence that our remote VMs are placed in groups with exceptionally high throughput links to the other groups, supported by the following reasoning:

$$p(\textit{greatest sample} \in \textit{percentile } t) \;=\; (1 - t^{\,o})$$

$$t = \sqrt[o]{1-p}$$

Where greatest sample is the largest sampled throughput among all scout observations, t is the percentile of all all links according to throughput, o= [10*S*(S-1)] is the number of scout observations, and p is our level of confidence

If we have a reasonably large job of 100 VMs and three scouts are used, this allows us to guarantee with 90% confidence that the first VM to migrate away from the main group(s) will be placed in the 96th percentile of desirable groups - in other words, the group with the best throughput.

This information populates the network table with viable options for placing VMs in areas of the data center that are known to have near optimal throughput, and allows us to avoid blind placement in a potentially slow area that would waste time. A heavily congested link may allow only a small fraction of the 10 Gb/s throughput that a non-congested route would permit. Although each scout requires about a 1% expenditure increase, in most cases the time savings from placing distributed VMs in desirable locations may increase the speed of the slowest transfer by an integer factor, perhaps even a full order of magnitude. In the case where a heavily congested route (as slow as ~50Mb/s in very high traffic) can be avoided, and remote VMs can be placed in a group that has the best average throughput (up to 10Gb/s) to the main group(s), such savings estimates are very reasonable.

6 Conclusion

This design marries the simplicity of a greedy algorithm with the advantages of taking easy network and space-availability measurements from the data center, to produce a placement which allows all VMs to finish quickly. While the placement may not be perfectly optimal, it is responsive to changes in the data center occupancy and network state and our algorithm is fast enough to recompute a near-optimal placement every second.

An interesting feature of our design is that it completely ignores the internal structure of the data center -- measuring individual bottleneck links with knowledge of the link structure would yield data far too complicated to be used by our greedy algorithm. Thus, this placement algorithm could work just as well on other data centers with dissimilar architectures, although some of the parameters (such as number of scouts and frequency of placement) may have to be adjusted to account for different data center sizes.

In fact, these same parameters could be adjusted even on this data center, and the overall time measured, to explore a completely different set of constraint values and numerically approximate still better parameters than the estimated ones presented above.