Group Allocator

Sanjeev Mehrotra

I. Introduction

A generic allocator attempts to match *requested resources*, R, to an optimal (in some sense) *allocatable resource location index*, $l_{opt} \in \{0, 1, \ldots, L-1\}$ with the location having *allocatable resources*, $A_{l_{opt}}$ and N_i being the name of the ith resource location. A_l are the allocatable resources available at allocatable location index l and L is the number of possible allocatable resource locations. If no location is found which satisfies the constraints for the requested resources, it sets $l_{opt} = -1$. Let $\mathcal{F} \subseteq \{N_0, N_1, \ldots, N_{L-1}\}$ be the set of feasible resource location names, and let s_l be a *score* at location $l \in \mathcal{F}$. Then,

$$l_{opt} = \begin{cases} \arg\max_{j \in \mathcal{F}} s_j & \text{if } \mathcal{F} \neq \emptyset \\ -1 & \text{if } \mathcal{F} = \emptyset \end{cases} \tag{1}$$

For example, in Figure 1, the allocator matches the given requested resources, R, to the allocatable resources at location 1.

We represent the requested resources as R, where R is a hash-map, i.e. a list of (K, V) key-value pairs, with a key `request_name` taken from a set, `requested_resources`. That is R[`request_name`] specifies the value needed to satisfy the request, with `request_name` $\in$ `requested_resources`. The exact function to used to determine the score and see if an allocatable resource location can satisfy the requested resource is completely arbitrary and can be dependent on the `request_name` or the allocatable resource being used to satisfy the request.

We represent the allocatable resources at location l as A_l, where A_l is a hash-map with a key `alloc_name` taken from a set of available resources at location l, given by `allocatable_resources_l`.

For a location l to satisfy the request, for each requested resource, R[`request_name`], there must be an allocatable resource, A_l[`alloc_name`] which satisfies the request. Let M_l be this mapping from requested resource to allocatable resource. That is M_l[`request_name`] is a value in the set `allocatable_resources_l`. If no allocatable resource specifies the request, then M_l[`request_name`] $=$ `nil`. Then, we can define the feasible set as the following.

$$\begin{aligned} \mathcal{F} = \{&N_l | l \in \{0, 1, \ldots, L-1\}, \\ &M_l[\texttt{request\_name}] \neq \texttt{nil} \quad \forall \texttt{request\_name} \in \texttt{requested\_resources}\}. \end{aligned} \tag{2}$$

We note that M_l[`request_name`] may have constraints, that is a given requested resource may only be satisfied by certain allocatable resources. For example, a CPU request can only be satisfied by an allocatable CPU resource, and not for example memory.

In the Kubernetes framework, such an allocator can be used to find the node to schedule a pod on, or can simply be used to see if resources fit on a given node by giving it one available resource location which is the node being tested to start with.

A. Example

As an example, R is the set of requested resources as in

```
cpu: 2
memory: 1GiB.
```

Here the set of `requested_resources` = {`cpu`, `memory`}, and thus `resource_name` can either be `cpu` or `memory`. For simplicity, we write this as $R =$`map[cpu: 2 memory: 1GiB]`.

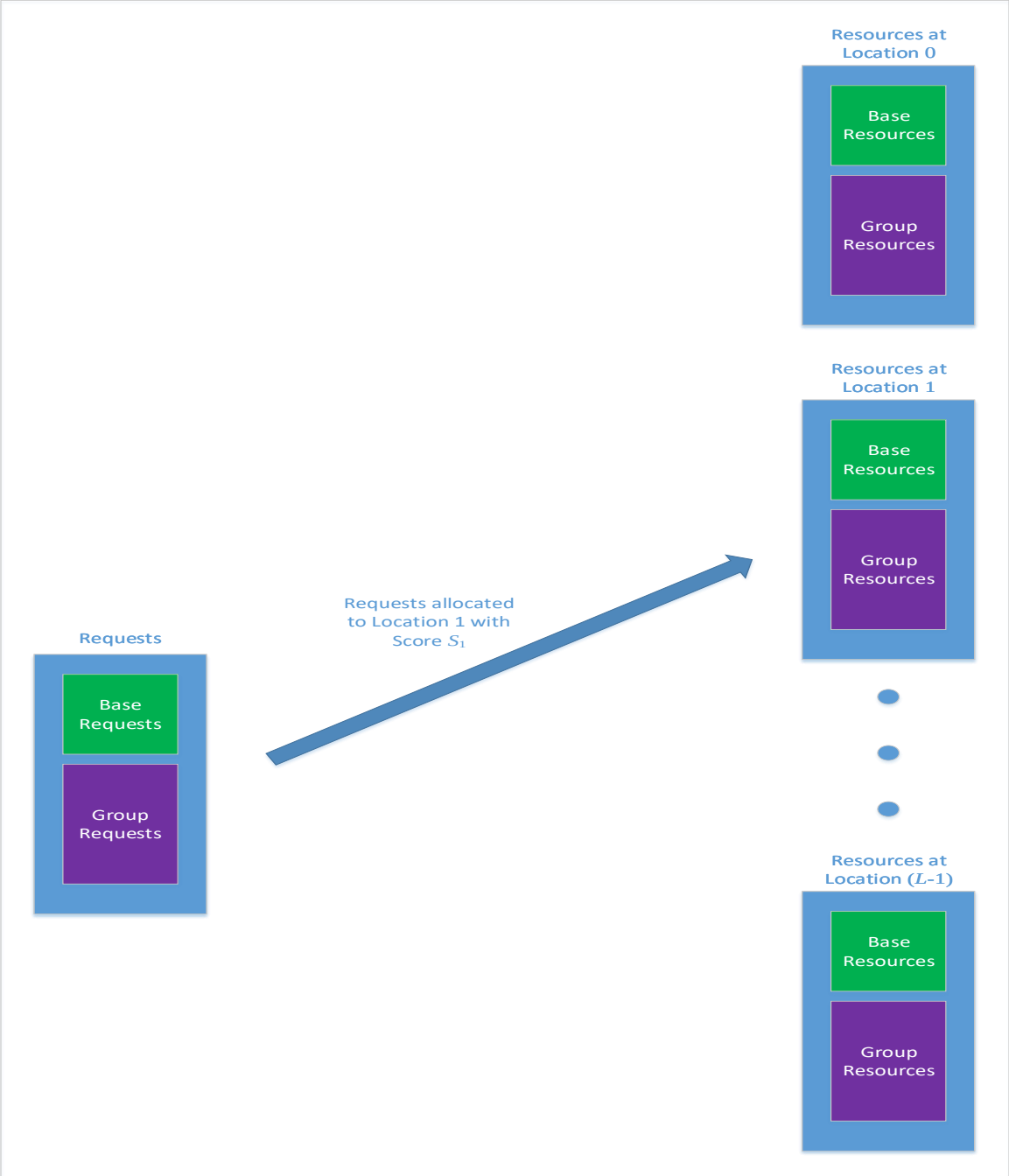

Fig. 1. Showing requests mapping to resources at given location

Similarly, A_l is the set of allocatable resources at location l. For example A_1 can be the set of allocatable resources at location 1 given by

```
cpu: 4
memory: 2GiB.
```

Here `allocatable_resources_1` = {`cpu`, `memory`}, and again `alloc_name` can either be `cpu` or `memory`. We can write A_1 =`map[cpu: 4 memory: 2GiB]`.

II. GROUP ALLOCATOR

In the simple case as given in Sec. I-A, the allocator can compare key by key to see if the requested resources R can fit onto allocatable resources at location l, A_l. For example compare requested `cpu` with allocatable `cpu` and similarly for `memory`. It does not make sense for a requested CPU resource to be satisfied by an allocatable memory resource. Thus, for any l, $M_l[\texttt{cpu}] = \texttt{cpu}$ if sufficient CPU is available, otherwise $M_l[\texttt{cpu}] = \texttt{nil}$. The same is true for memory. Having only such resources makes the job of the allocator fairly straightforward. Thus for simple cases like this, the mapping, M_l is not something that is found by the allocator, but rather a static identity map.

However, this becomes infeasible for more complicated scenarios. As an example, consider GPU resources, where a single location can have multiple GPUs, each with differing memory and similarly, the request can require a number of GPUs with differing memory requirements. Although it may seem difficult to express such constraints with simple key-value pairs for the requested resources and allocatable resources, it is fairly doable. For example, one way to do this is to express requested resources and allocatable resources for each GPU *separately*. Then, we can write requested resources, R, as

```
cpu: 2
memory: 1GiB
gpu/0/cards: 1
gpu/0/memory: 1GiB
gpu/1/cards: 1
gpu/1/memory: 2GiB,
```

where we have requested 1 GPU with minimum 1GiB memory and another with minimum 2GiB memory.

Similarly, a resource location can advertise allocatable resources, A_l, as

```
cpu: 4
memory: 2GiB
gpu/dev0/cards: 1
gpu/dev0/memory: 1GiB
gpu/dev1/cards: 1
gpu/dev1/memory: 3GiB
gpu/dev2/cards: 1
gpu/dev2/memory: 2GiB
gpu/dev3/cards: 1
gpu/dev3/memory: 4GiB,
```

where we have 4 GPUs with differing memory on the node.

In such a case, a simple allocator cannot just look for a resource with the same key as that being requested. For example, it is clear that the request can be satisfied by the resource location. However, there is no key `gpu/0/cards` present in the resource location even though it is a requested key. This is the main need for what we define as the *group allocator*.

The job of the group allocator becomes the following.

- Find the best allocatable resource location index(in some sense), l_{opt}, amongst the feasible set, $\mathcal{F}$.
- Obtain the mapping from requested resource to allocatable resource on for that best allocatable resource location, $M_{l_{opt}}$.

In the process of determining l_{opt}, the allocator may compute a score for each location to find the best one.

The second item is needed as the map will not be the identity map anymore. An example of such a mapping, $M_{l_{opt}}$, could be the following.

```
cpu: cpu
memory: memory
gpu/0/cards: gpu/dev0/cards
gpu/0/memory: gpu/dev0/memory
```

```
gpu/1/cards: gpu/dev2/cards
gpu/1/memory: gpu/dev2/memory
```

A. Partitioning

The first task of the group allocator is partitioning. Both the requested resources as well as the allocatable resources of a given location are partitioned into "base" and "group" resources. A base resource is one which can directly be compared by looking at the key. For example, a base requested resource requires a base allocatable resource with the same key, for example, in the example above `cpu` and `memory`. That is for a base resource

$$M_l[\texttt{request\_name}] = \texttt{request\_name}. \tag{3}$$

A "group" resource on the other hand, may match a requested resource to an allocatable resource with a differing key.

In our framework, all requested resources which are of the format,

$$\texttt{request\_name} = \texttt{request\_groupname/request\_groupindex/request\_basename} \tag{4}$$

are considered to be "group" requested resources by the allocator, where the keys are in the sets,

$$\texttt{request\_groupname} \in \texttt{request\_subgroupnames} \tag{5}$$

$$\texttt{request\_groupindex} \in \texttt{request\_subgroupindices[request\_groupname]} \tag{6}$$

$$\texttt{request\_basename} \in \texttt{request\_resourcebasenames[request\_groupname][request\_groupindex]} \tag{7}$$

Similarly, all allocatable resource which are of the format,

$$\texttt{alloc\_name} = \texttt{alloc\_groupname/alloc\_groupindex/alloc\_basename} \tag{8}$$

are "group" allocatable resources, with the keys in the sets

$$\texttt{alloc\_groupname} \in \texttt{alloc\_subgroupnames} \tag{9}$$

$$\texttt{alloc\_groupindex} \in \texttt{alloc\_subgroupindices[alloc\_groupname]} \tag{10}$$

$$\texttt{alloc\_basename} \in \texttt{alloc\_resourcebasenames[alloc\_groupname][alloc\_groupindex]} \tag{11}$$

Since for base resources, $M_l[\texttt{request\_name}] = \texttt{request\_name}$, it is sufficient to look at $A_l[\texttt{request\_name}]$ and compare. If `request_name` does not exist in the set of `allocatable_resources`, then the resource request cannot be satisfied.

To check if a group resource being requested can fit onto a location with allocatable resources, group resources are first organized into subgroups. A new subgroup is formed for each unique combination of (`request_groupname, request_groupindex`), as

$$R_G[\texttt{request\_groupname}][\texttt{request\_groupindex}][\texttt{request\_basename}] = R[\texttt{request\_groupname/request\_groupindex/request\_basename}]. \tag{12}$$

Similarly, the group allocatable resources at location l which are of the format A_l[`alloc_groupname`] [`alloc_groupindex`] [`alloc_basename`] are partitioned, resulting in a new subgroup for each combination of (`alloc_groupname, alloc_groupindex`),

$$A_{lG}[\texttt{alloc\_groupname}][\texttt{alloc\_groupindex}][\texttt{alloc\_basename}] = A_l[\texttt{request\_groupname/request\_groupindex/request\_basename}]. \tag{13}$$

1) Example: For example, in the example above, the requested resources produces two requested subgroups $R_G[\texttt{gpu}][0]$ and $R_G[\texttt{gpu}][1]$, where $R_G[\texttt{gpu}][0]$ is given by

```
cards: 1
memory: 1GiB,
```

and $R_G[\texttt{gpu}][1]$ is given by

```
cards: 1
memory: 2GiB.
```

Similarly, the allocatable resources at the location 1 produce four allocatable subgroups, $A_{1G}[\texttt{gpu}][\texttt{dev0}]$, given by

```
cards: 1
memory: 1GiB,
```

$A_{1G}[\texttt{gpu}][\texttt{dev1}]$ given by,

```
cards: 1
memory: 2GiB,
```

$A_{1G}[\texttt{gpu}][\texttt{dev2}]$, given by

```
cards: 1
memory: 2GiB,
```

and $A_{1G}[\texttt{gpu}][\texttt{dev3}]$, given by

```
cards: 1
memory: 4GiB.
```

B. Allocation

Now, the group allocator is recursively run on each of the requested subgroups, $R_G[\texttt{request\_groupname}][\texttt{request\_groupindex}]$. For each requested subgroup, all allocatable subgroups where `alloc_groupname` = `request_groupname` are considered as possible resource locations. The number of possible resource locations for the allocation of a subgroup onto resource location l is given by

$$L_{lG}[\texttt{request\_groupname}] = \text{length}(A_{lG}[\texttt{request\_groupname}]). \tag{14}$$

That is it is from the set `alloc_subgroupindices[request_groupname]`. Let's call this location $n_{lG,opt}[\texttt{request\_groupname}][\texttt{request\_groupindex}]$. For simplicity, we can simply denote it as $n_{lG,opt}$.

In addition, the recursive call to the allocator returns the mapping for the subgroups M_{lG}. The mapping returned by the recursive call to the group allocator for the subgroups can be used to get back the mapping for the group via the following,

$$\begin{aligned} M_l[\texttt{request\_groupname/request\_groupindex/request\_basename}] = \\ \texttt{request\_groupname}/ \\ \texttt{alloc\_subgroupindices}[n_{lG,opt}]/ \\ M_{lG}[\texttt{request\_groupname}][\texttt{request\_groupindex}][\texttt{request\_basename}] \end{aligned} \tag{15}$$

If any of the subgroup allocations fail at location l (i.e., there is no feasible location for the subgroup), then the requested resources cannot fit at location l.

If all the subgroup allocation succeed, and if the base resource allocations are deemed to be sufficient, then a score for location l can be computed to be used by the allocator.

1) Example: For example, when considering the fit for location 1 using the allocatable resource A_1, for request R, the group allocator does the following.

- Compare the base resources `cpu` and `memory` to see if they fit.
- Recursively call the group allocator to see if R_G[gpu][0] can fit onto any of the allocatable resources subgroups, $\{A_{1G}$[gpu][dev0], A_{1G}[gpu][dev1], A_{1G}[gpu][dev2], A_{1G}[gpu][dev3]$\}$. The group allocator returns $(fit, mapping)$. Let M_{1G}[gpu][0] be the mapping and let n_{1G}[gpu][0] $\in \{-1, 0, 1, 2, 3\}$ be the optimal location index. -1 specifies that no location was found out of the four locations.
- Recursively call the group allocator to see if R_G[gpu][1] can fit onto any of the allocatable resources subgroups, $\{A_{1G}$[gpu][dev0], A_{1G}[gpu][dev1], A_{1G}[gpu][dev2], A_{1G}[gpu][dev3]$\}$. The group allocator returns $(fit, mapping)$. Let M_{1G}[gpu][1] be the mapping and let n_{1G}[gpu][1] $\in \{-1, 0, 1, 2, 3\}$ be the optimal location index. -1 specifies that no location was found out of the four locations.
- If the base resources fit, and the recursive calls to the group allocator return a fit (i.e. a location not equal to -1), then the requested resources can fit.
- From Eqn. 15, obtain the final mapping using the following.

  cpu: cpu
  memory: memory
  gpu/0/cards: gpu/alloc_subgroupindices[$n_{1G,opt}$[gpu][0]]/cards
  gpu/0/memory: gpu/alloc_subgroupindices[$n_{1G,opt}$[gpu][0]]/memory
  gpu/1/cards: gpu/alloc_subgroupindices[$n_{1G,opt}$[gpu][1]]/cards
  gpu/1/memory: gpu/alloc_subgroupindices[$n_{1G,opt}$[gpu][1]]/memory,

  where $n_{1G,opt}$[gpu][request_groupindex] is the location of the allocatable subgroup with `alloc_name=gpu` returned by the recursive call to the group allocator. Note that `alloc_subgroupindices`[$n_{1G,opt}$[gpu][*]] will be from the set {dev0, dev1, dev2, dev3}.
- If a fit is found, then a score is computed for location 1.

For example, suppose $n_{1G,opt}$[gpu][0] $= 0$ and $n_{1G,opt}$[gpu][1] $= 2$, then `alloc_subgroupindices` [$n_{1G,opt}$[gpu][0]] = dev0 and `alloc_subgroupindices`[$n_{1G,opt}$[gpu][1]] = dev2. The recursive call to the group allocator returns the mapping for both subgroups, as M_{1G}[gpu][0] and M_{1G} [gpu][1], which are both given by,

cards : cards
memory : memory.

Then, using Eqn. 15, we get the following mapping

cpu: cpu
memory: memory
gpu/0/cards: gpu/dev0/cards
gpu/0/memory: gpu/dev0/memory
gpu/1/cards: gpu/dev2/cards
gpu/1/memory: gpu/dev2/memory,

In the final mapping, for base resources, M[resource] = resource. For group resources, the request_groupindex for the requested resource is replaced by `alloc_subgroupindices`[$n_{1G,opt}$ [request_groupname][request_groupindex]], where $n_{1G,opt}$ is the best location for the subgroup amongst the allocatable subgroups with the same groupname.

C. Scorer

One component which can be used to find the best location amongst a list of allocatable resource locations is a scorer. For a location l, it uses allocatable resources, A_l, and resources already being used along with the requested resources, R, and requested resource mapping M to find a score for the fit. We define s_l to be the score for the location l. Let U_l be the resources being used at location l. It is a hash-map with the same keys used to index A_l, that is it is indexed using `alloc_name`.

We define s_l to be the sum of scores over all allocatable resources, that is

$$RL_l[\texttt{alloc\_name}] = \{R[\texttt{request\_name}] : M_l[\texttt{request\_name}] = \texttt{alloc\_name}\} \quad (16)$$

$$s_l = \sum_{\texttt{alloc\_name} \in \texttt{allocatable\_resources\_l}} S_{\texttt{alloc\_name}}(A_l[\texttt{alloc\_name}], U_l[\texttt{alloc\_name}], RL_l[\texttt{alloc\_name}]) \quad (17)$$

that is the scorer function $S_{\texttt{alloc\_name}}$ itself is a function of the resource being allocated upon, and it is a function of the allocatable resource A_l for the given resource, the amount of used resource U_l for the given resource, and the list of requests R which map to using the given resource. In addition to a score, the scorer can also tell whether a sufficient amount of allocatable resource exists in order to satisfy the request. For example, the scorer could return -1 to denote that sufficient resource does not exist. In practice, the first step the scorer will do is find a list of resources being mapped to a given allocatable resource, RL, as in Eqn. 16. The list RL is then passed to the scorer.

In addition to computing the score, an arbitrary function, $\Phi_{\texttt{alloc\_name}}$, can be used to update the used resources for an allocatable resource, $U_l[\texttt{alloc\_name}]$, using

$$U'_l[\texttt{alloc\_name}] = \Phi_{\texttt{alloc\_name}}(A_l[\texttt{alloc\_name}], U_l[\texttt{alloc\_name}], RL_l[\texttt{alloc\_name}]), \quad (18)$$

where U'_l is used to denote the updated used resources.

Note that the scorer function, S, and used resource updater, Φ, can also be dependent upon the request being made, and can also have additional state that may be needed. As an example, in Kubernetes, pods consist of both initialization containers and running containers. Initialization containers are first run sequentially (one-by-one), and once all initialization containers complete, then all running containers are run in parallel. Here, the scorer itself is dependent upon whether the request is being made on behalf of a running container or an initialization container. In addition, additional state such as the resource being requested by the pod can be used as additional state for the scorer. Thus in an actual implementation of the scorer function, additional parameters may be passed in.

As an example, for linearly consumable resources, we can use the following for the resource updater and scorer. The resource updater and scorer must run in a particular order. When resources are being consumed, all running containers must be updated and scored first, followed by initialization containers. Let $P[\texttt{alloc\_name}]$ be the resources being consumed by a pod. Then, we first run the following for running containers.

$$P'_l[\texttt{alloc\_name}] = P_l[\texttt{alloc\_name}] + RL_l[\texttt{alloc\_name}] \quad (19)$$

$$U'_l[\texttt{alloc\_name}] = U_l[\texttt{alloc\_name}] + RL_l[\texttt{alloc\_name}] \quad (20)$$

$$s_l = \frac{U'_l[\texttt{alloc\_name}]}{A_l[\texttt{alloc\_name}]} \quad (21)$$

Then, for initialization containers, we do the following

$$P'_l[\texttt{alloc\_name}] = \max(P_l[\texttt{alloc\_name}], RL_l[\texttt{alloc\_name}]) \quad (22)$$

$$U'_l[\texttt{alloc\_name}] = U_l[\texttt{alloc\_name}] + (P'_l[\texttt{alloc\_name}] - P_l[\texttt{alloc\_name}]) \quad (23)$$

$$s_l = \frac{U'_l[\texttt{alloc\_name}]}{A_l[\texttt{alloc\_name}]} \quad (24)$$

We see that for initialization containers, additional state regarding the amount of resources being used by running containers for the pod is used as an additional variable to modify the function Φ. If we put all equations together, we get the following for the scorer,

$$s_l = \frac{U_l[\texttt{alloc\_name}] + \max(P_l[\texttt{alloc\_name}], RL_l[\texttt{alloc\_name}]) - P_l[\texttt{alloc\_name}]}{A_l[\texttt{alloc\_name}]}, \quad (25)$$

and thus it is also a function of current pod usage.

When a pod is being removed from a node, then the entire pod usage is first computed and then subtracted from the node usage.

III. Implementation in Kubernetes

The group allocator is implemented in Kubernetes in the file `plugin/pkg/scheduler/algorithm/ predicates/grpallocate.go`. All requested resources as well as allocatable resources on a node with the prefix `alpha.kubernetes.io/group-resource` are considered to be "group" resources. The type `GrpAllocator` is the main class which performs the group allocator and stores state.

The group allocator's job is two-fold, one to find a suitable location for the requested resources amongst the *resource locations* with *allocatable resources* and two to find a mapping between requested resources and allocatable resources on the resource location.

Initially, we consider there to be only one resource location, which is the node group resources. So the initial group allocator call attempts to find whether the node is able to satisfy the requests and if it can satisfy the requests, then it also outputs a mapping between requests and allocatable resources on the node.

The variables `RequiredResource` and `AllocResource` correspond to key-value pairs with keys being strings and value being int64 corresponding to the required and allocatable resources at a given node respectively. These correspond to resources at the top-level group allocator.

The actual required resource and allocatable resource for a given group are specified in `GrpRequiredResource` and `GrpAllocResource` which are string to string mappings. Therefore, `GrpRequiredResource map[string]string` is a map whose keys specify the names of the requested resources. And, `RequiredResources[GrpRequiredResource[resource_key]]` is the amount of requested resource for with the name `resource_key`. In addition, `GrpAllocResource map[string](map[string]string)` is a map whose keys specify locations for resource locations. And, `GrpAllocResource[resource_location_key]` specifies a list whose keys specify the resources available at the allocatable location `resource_location_key`. And `AllocResource[GrpAllocResource[resource_location_key][resource_key]]` specifies the amount of allocatable resource at resource location `resource_location_key` with the name `resource_key`.

For simplicity all variables in the group allocator are specified with respect to the top level group. The actual ones for a given group are simply found by first looking up in `GrpRequiredResource` or `GrpAllocResource` to find the actual key for a given group.

The group allocator consists of several core functions.

- `allocateGroup` allocates requested resources amongst a list of resource locations. It then picks a location with the best score amongst the list of feasible locations, provided one exists.
- `allocateGroupAt` attempts to allocate the request to a particular location and if it fits, it finds a score, along with a requested resource to allocatable resource mapping.
- `resourceAvailable` checks if sufficient *base resources* are available at the resource location.
- `allocateSubGroups` performs recursive calls to allocateGroup for all subgroups, if any exist.
- `findScoreAndUpdate` performs scoring for the allocation and updates used resources.

The scorer is a function with signature,

```
func(allocatableResource int64, usedByPod int64, usedByNode int64,
requested []int64, initContainer bool) (found bool, score float64,
usedByContainer int64, newUsedByPod, newUsedByNode int64)
```

It is specified for all allocatable resources at a given node. A different function may be used to determine fit if desired.

IV. GPU Affinity Example

Here we provide an example of using the group allocator for allocating GPUs with affinity. GPUs are often connected via a high speed peer-to-peer link, for example, NVLink in NVIDIA GPUs with Pascal

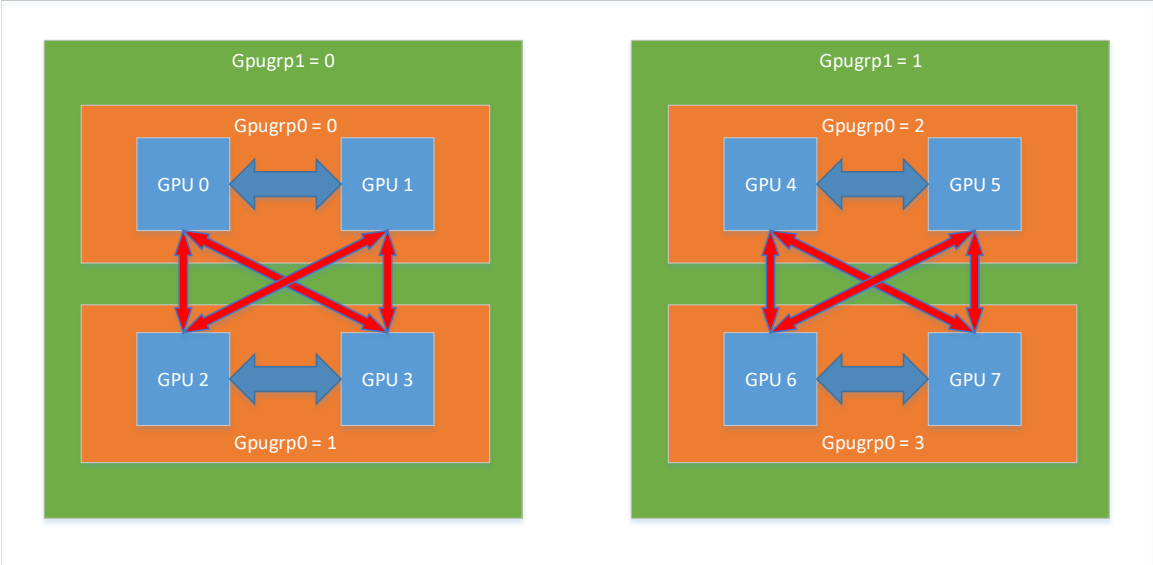

Fig. 2. GPU with NVLink and corresponding GPU groups, showing both high speed and lower speed interconnects.

architecture as shown in Fig. 2. GPU0 is connected via high-speed interconnect to GPU1 and a lower speed interconnect to GPU2 and GPU3.

Then, one way to specify allocatable resources is the following.

```
Gpugrp1/0/Gpugrp0/0/gpu/dev0/cards: 1
Gpugrp1/0/Gpugrp0/0/gpu/dev1/cards: 1
Gpugrp1/0/Gpugrp0/1/gpu/dev2/cards: 1
Gpugrp1/0/Gpugrp0/1/gpu/dev3/cards: 1
Gpugrp1/1/Gpugrp0/2/gpu/dev4/cards: 1
Gpugrp1/1/Gpugrp0/2/gpu/dev5/cards: 1
Gpugrp1/1/Gpugrp0/3/gpu/dev6/cards: 1
Gpugrp1/1/Gpugrp0/3/gpu/dev7/cards: 1
```

That is, two GPUs with high-speed interconnect are first grouped into Gpugrp0, with indices $\{0, 1, 2, 3\}$. Then, two successive groups of Gpugrp0 are grouped into Gpugrp1 which has the lower speed interconnect, with indices $\{0, 1\}$.

Requested resources can be similarly grouped depending on which interconnects are required. For example,

```
Gpugrp1/R0/Gpugrp0/RA/gpu/gpu0/cards: 1
Gpugrp1/R0/Gpugrp0/RA/gpu/gpu1/cards: 1
Gpugrp1/R1/Gpugrp0/RA/gpu/gpu2/cards: 1
Gpugrp1/R1/Gpugrp0/RA/gpu/gpu3/cards: 1
Gpugrp1/R1/Gpugrp0/RB/gpu/gpu4/cards: 1
Gpugrp1/R1/Gpugrp0/RB/gpu/gpu5/cards: 1
```

The top-level group allocator will make two recursive calls to group allocate `request_groupname = Gpugrp1` with `request_groupindex = 0` and `request_groupindex = 1`.

The call to allocate `request_groupname = Gpugrp1` for `request_groupindex = 0` will then make a single recursive call to group allocate `request_groupname = Gpugrp0` with `request_groupindex = A`. This in turn will make two recursive calls to group allocate `request_groupname = gpu` with `request_groupindex = dev0` and another for `request_groupname = gpu` with `request_groupindex = dev1`. Each of these will have the base resource of `cards`.

The call to allocate `request_groupname = Gpugrp1` for `request_groupindex = 1` will then make two recursive call to group allocate, one for `request_groupname = Gpugrp0` with `request_groupindex = A` and another for `request_groupname = Gpugrp0` with `request_groupindex = B`. Each of these two will make two more recursive calls.

The final allocations to perform will be the following.

```
<TopLevel>
  Gpugrp1/R0
    Gpugrp1/R0/Gpugrp0/RA
      Gpugrp1/R0/Gpugrp0/RA/gpu/gpu0
        Gpugrp1/R0/Gpugrp0/RA/gpu/gpu0/cards -- base resource
      Gpugrp1/R0/Gpugrp0/RA/gpu/gpu1
        Gpugrp1/R0/Gpugrp0/RA/gpu/gpu1/cards -- base resource
  Gpugrp1/R1
    Gpugrp1/R1/Gpugrp0/RA
      Gpugrp1/R1/Gpugrp0/RA/gpu/gpu2
        Gpugrp1/R1/Gpugrp0/RA/gpu/gpu2/cards -- base resource
      Gpugrp1/R1/Gpugrp0/RA/gpu/gpu3
        Gpugrp1/R1/Gpugrp0/RA/gpu/gpu3/cards -- base resource
    Gpugrp1/R1/Gpugrp0/RB
      Gpugrp1/R1/Gpugrp0/RB/gpu/gpu4
        Gpugrp1/R1/Gpugrp0/RB/gpu/gpu4/cards -- base resource
      Gpugrp1/R1/Gpugrp0/RB/gpu/gpu5
        Gpugrp1/R1/Gpugrp0/RB/gpu/gpu5/cards -- base resource
```

Of course, each allocation will attempt to search for multiple locations for the fit. A diagram showing the call is shown in Fig. 3. The green circles show group allocators. The blue boxes show the allocatable locations being searched. Each blue box will perform a search for all resources which are considered as "base resources" for that level. Each blue box may have further subgroups to search which will be shown as descendants of the blue box. If there are no group resources left (i.e. all resources are base resources), then the box will have no descendants. For example, the leaf of the tree at the top right of the figure is requested subgroup `Grp1/R0/Grp0/RA/gpu/0` checking if resources fit onto `Grp1/0/Grp0/0/gpu/dev0`. The only requested resource is `cards` which is checked. Each group allocator (green circle) picks the best of all locations. That is it picks the best blue box which descends from the node. In the figure, for simplicity, only the `Grp1/R0` branch is shown. Another branch for `Grp1/R1` will also be present which is not shown in the figure.

V. Requested Resource Translation

For simplicity, requests may specified in the lower groups without specifying the higher group enumeration. For example, if we need two GPUs within the same `Gpugrp0`, we can simply specify the request as

```
Gpugrp0/RA/gpu/gpu0/cards: 1
Gpugrp0/RA/gpu/gpu1/cards: 1
```

If the resources being advertised also include a `Gpugrp1`, then the requested resources will automatically get assigned to a unique `Gpugrp1`, a different one for each differing `Gpugrp0`.

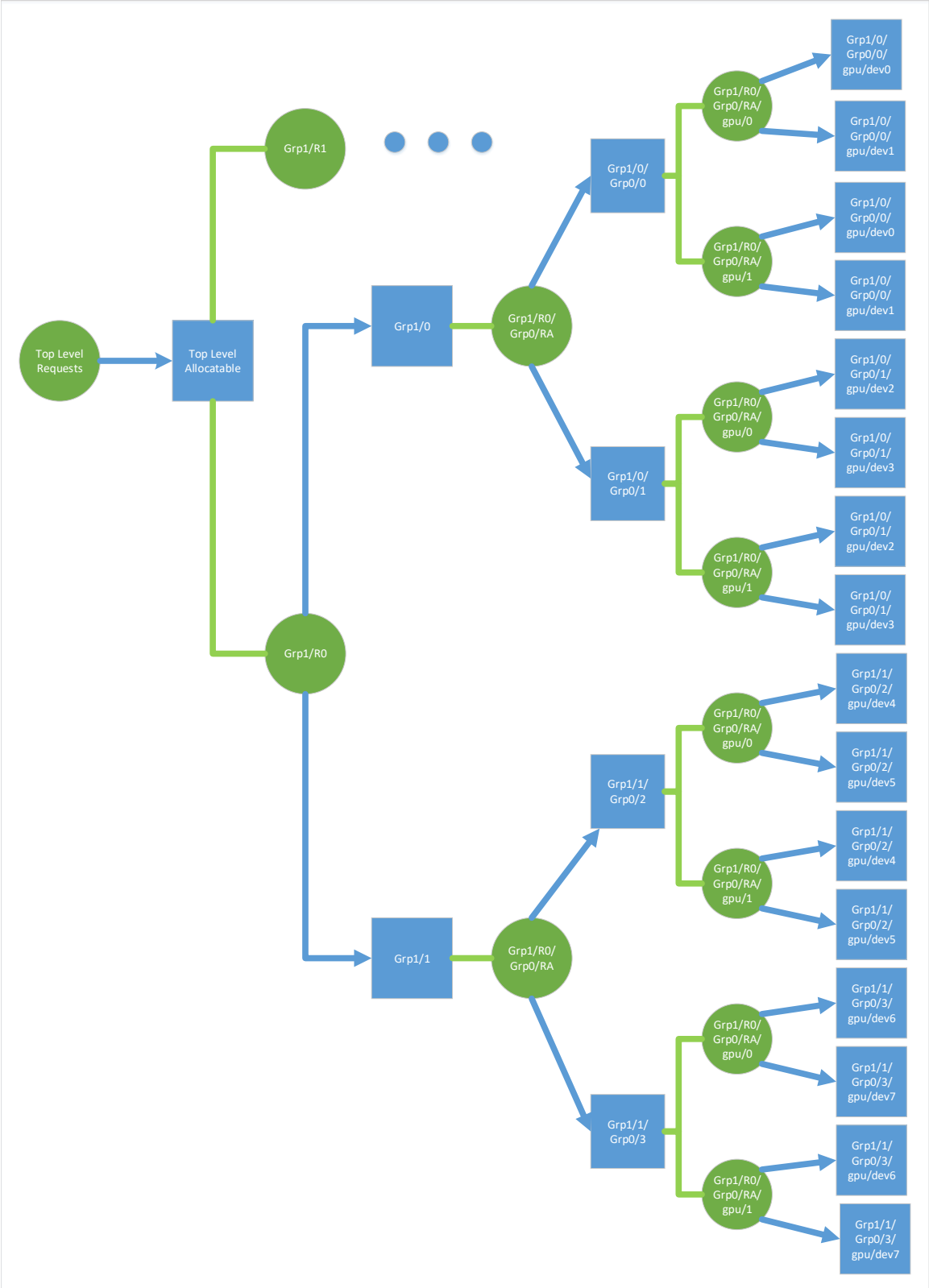

Fig. 3. Figure showing the group allocators and the locations being searched. For simplicity, “Gpugrp” has simply been written as “Grp”. The branch for `Grp1/R1` is not shown.