RESOURCE MANAGEMENT IN LARGE-SCALE DATA CENTERS

Introduction

Data analytics performed on large volumes of data (for social networks, recommendation systems, scientific computing, etc.)

Data centers must *scale up* to adapt to the growing size of data sets.

Cloud computing enables the customers to run large applications and store large amounts of data without concern about the configuration of hardware and software infrastructures.

In reality **clouds** are large *sets of* computing *clusters*.

A **cluster** is a set of computers that *may be distant*, connected via a network.
These computers act as a single efficient machine characterized by its resource capacities: memory, CPU, Disk I/O and network bandwidth.

Data sets are partitioned and stored in data center nodes, and
computing is *parallelized* to gain in computing speed and efficiency.

Adapted computing environments, called *data analytics* ***frameworks***, offer the users
- *platform* and *services* to host their applications an develop new applications, and
- Application Programming Interfaces (*APIs*).

Best known frameworks are *MapReduce*, *Dryad*, *Pregel* and *Apache Yarn*.

Frameworks ensure ***data storage*** and ***application execution*** on data center nodes,
by applying an automatic and logic division of data and jobs into fine grained ***blocks*** and ***tasks***.

A job's *input file* is partitioned according to the File Management System (FMS) into *blocks*.

To execute *jobs* on the nodes, the jobs are partitioned into fine grained *tasks*.

Each block of data is stored in a specific node
hence the input data of a given task could be on one or many nodes.

The system assigns one or more slots per task.

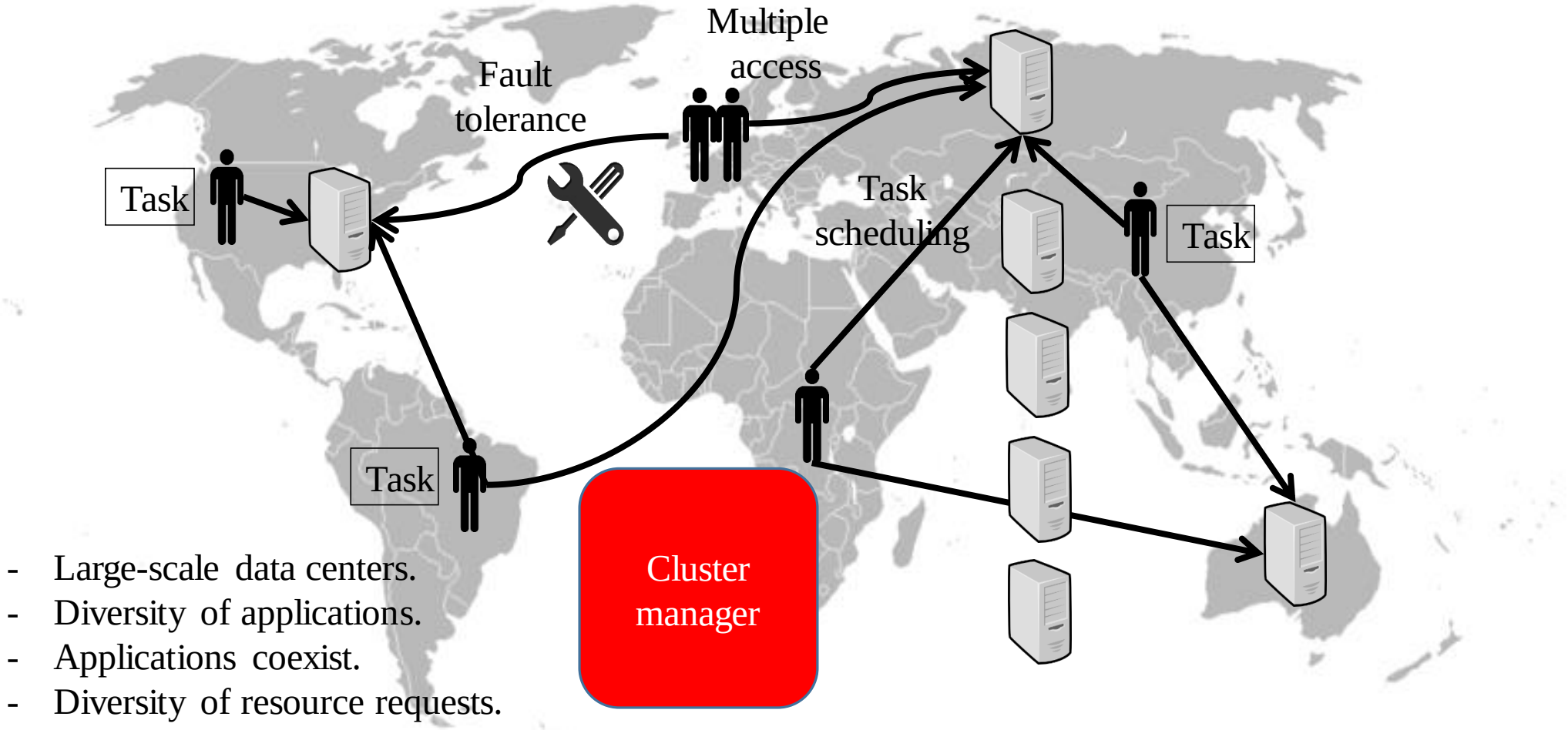
Term	Meaning
block	atomic unit of storage by the distributed file system
file	collection of blocks
workflow	directed acyclic graph indicating how data flows between phases
job	an execution of the workflow
task	atomic unit of computation with a fixed input
slot	computational resources allotted to a task on a machine

Terminology used in data analytics frameworks

Context

In such large-scale data centers, many frameworks are running *competitively* on the same cluster (each framework aims at launching its tasks independently of the other ones).

Problem: How to share the common physical infrastructure between the frameworks while ensuring a fair share to everyone and optimizing resource utilization.

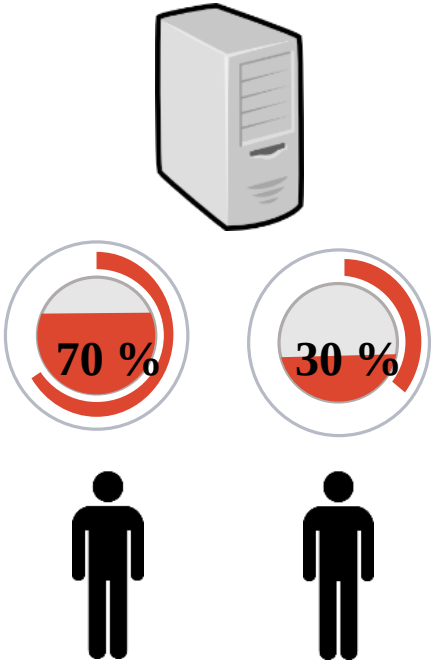


- Large-scale data centers.
- Diversity of applications.
- Applications coexist.
- Diversity of resource requests.

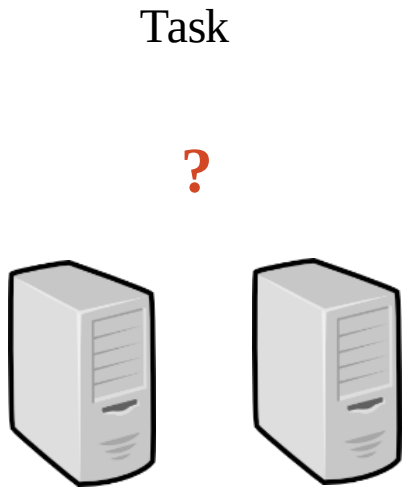
Two classes of resource management problems:
— the tasks are known at the beginning (*off-line*).
— the tasks arrive progressively (*on-line*), hence the needs in resources of the applications cannot be determined beforehand.

In actual clusters where hundreds of applications are simultaneously running and continuously generating tasks.
The associated resource management problem belongs to the *on-line* class.

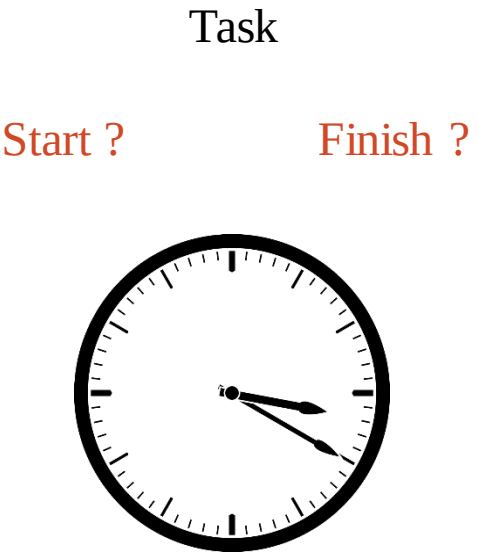
Scheduling multiple types of jobs on heterogeneous machines is NP-Hard.
Need heuristics that will estimate long-term task behavior based on actual resource requests.



1 **Resource Sharing**



2 **Task placement**



3 **Task scheduling**

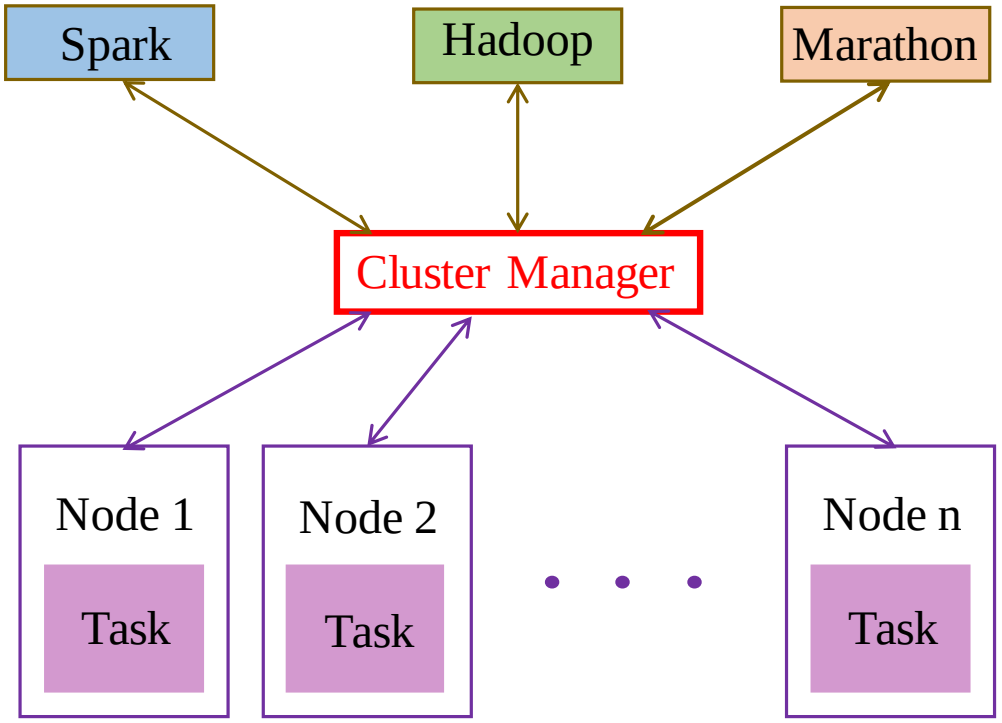
Resource providers must satisfy clients' specifications and requirements defined in *Service-Level Agreements* (*SLAs*), which are parts of contracts between service provider and service user in which quality of service and responsibilities are defined.

Trying to attain multiple objectives makes resource management particularly difficult:

— ***User Objectives***: *successfully* run the maximum number of tasks while client specified constraints are respected (by the provider).
One such constraint may be to run two tasks on two different servers or racks (groups of servers of similar characteristics and generally near to each other).

— ***Provider Objectives***: not only satisfy the client but also optimize resource utilization and maximize the number of tasks accepted (or minimize rejection rate).
Thus the provider would like
- to decrease the number of servers ON, in order to reduce energy consumption, and
- to assign to these servers (and run) as many tasks as possible.

To achieve these goals many *cluster managers* have a *hierarchical 2-level* architecture, effecting a concrete separation between *frameworks* and *data centers physical nodes*.
Apache Mesos and *Yarn* are examples of such cluster managers.
They orchestrate the communication between the frameworks and the servers.



Integration of cluster manager into cluster's architecture

A framework submits its requests to the cluster manager, which has an instant vision of the data center resources.

The cluster manager, which is the only entity which can get the amount of free/used resources in the cluster, responds with the availability of its machines.

The framework decides whether to accept or to reject the manager's offer.
Once it has accepted an offer, it sends its tasks to be run on the offered nodes.

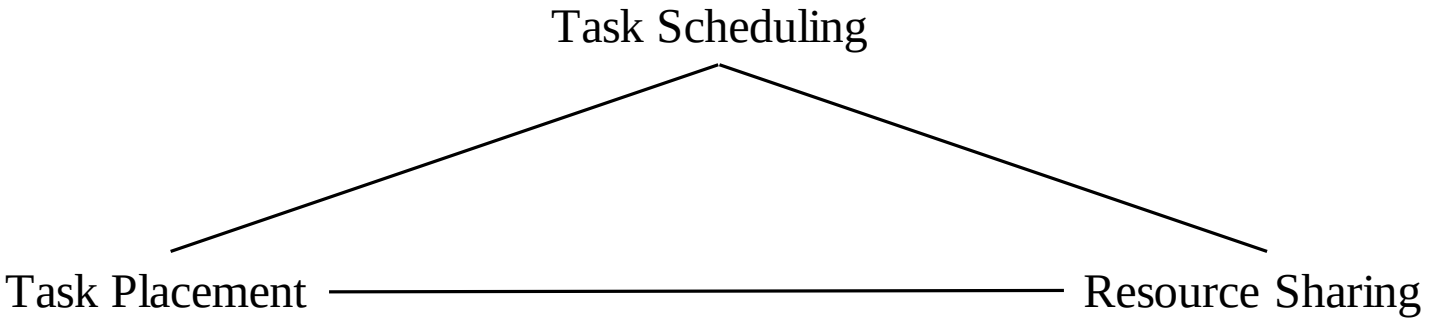
Cluster management

Many cluster managers have been developed which centralize cluster management on *one orchestrator node* in order to improve the efficiency of management and supervision.

These cluster managers have many roles:
schedule tasks on a cluster, control access to shared resources, manage task submission and monitoring, and coordinate the cluster's fault tolerance mechanisms.

We will concentrate on task scheduling, task placement and resource sharing.

In a hierarchical 2-level cluster manager, the *resource sharing* policy defines the way the resources can be shared between the frameworks that are in concurrent access within a data center.



High dependence between task scheduling, placement and resource sharing.

Task scheduling determines the start and finish time of a task.
Scheduling complexity increases with:

- *time constraints* (a task must finish at a specific date), or
- *resource constraints* (in order to be run, a task requires a specific amount of resources in memory, CPU, disk and network bandwidth),
- *environment heterogeneity*, with multiple users and varied demands.

Efficient scheduling algorithms already exist for both centralized and distributed systems. In a cloud infrastructure, in which big data jobs are running on large-scale clusters, scheduling requires developing new strategies because of the new constraints imposed by the environment itself.

In addition to resource constraints, *task placement constraints* should also be considered in large-scale data centers.

Task placement is deciding on which machine a task should be executed.
Taking this decision becomes complex in a multi-resource environment with heterogeneous applications and demands.

Considering task placement, ***constraints*** will delay tasks, yet they must be considered because of three reasons:

1. *Heterogeneous machines*: in a data center, machines often have different hardware characteristics and, in a large data center, machine homogeneity is impossible because of financial costs.
2. *Application optimization*: choose a specific machine to run a specific task based on a compromise in resource utilization.
3. *Problem avoidance*: prevent a problem that could be caused by a bad choice of machine such as an execution error due to machine clock speed.

In addition to task scheduling and placement, *resource sharing* must also be considered, making cluster management particularly complex.

Resource sharing consists in defining a specific amount of each resource (memory, CPU, disk and bandwidth) to give to each *application* running on the data center. The problem related to such decisions concerns ***fairness*** *across frameworks*.

Fairness is an abstraction of an *equitable distribution of something* between agents, users or applications; it means giving each entity an *appropriate* amount of the shared object.

Defining a fair share between concurrent entities requires considering many constraints regarding the entities and the surrounding environment.
In cloud infrastructures, sharing the data center's servers concerns many users and multiple resources, thus complex mechanisms must be established to ensure fairness.

Three questions must be asked when deploying any allocation strategy in any system:
— **Q1** : What is fairness?
— **Q2** : How do we measure the fairness of a system with regard to all the individuals?
— **Q3** : How to make a system fair?

Max-min fairness provides an answer**.**
An allocation is ***max-min fair*** if:
it is feasible (no constraint, especially capacity, is broken), and
an increase of allocation to one entity can be achieved only by decreasing the allocation of another entity with an equal or lower allocation.

Task scheduling

Max-min fairness has been widely deployed to perform ***task scheduling***.

A variant of this scheduling policy is *weighted max-min fairness*, which gives weights to the users that determine a priority order between them.

Task placement

In a cluster, (weighted) ***max-min fairness*** is implemented in the cluster manager to ***assign workflow tasks*** to the appropriate resource(s).

Among the algorithms proposed to implement weighted max-min fairness are *round robin* and *weighted fair queuing*.

Max-min fairness sorts resources by their increasing demands, it ensures that a task cannot get more than its demand and that tasks with satisfied demands get an equal share of the resource.

Max-min fairness proved its *effectiveness* as a ***fair resource allocation policy*** and its ability to isolate resource demands only *when a single resource is shared***.**

However, applications running on large data sets have diverse demands on more than one resource.

Resource sharing

In large-scale clusters, applications coexist and worse, they require multi-resources to accomplish their tasks. Thus sharing resources between applications is quite complex.

In large-scale data centers, there is an unprecedented heterogeneity in workload specifications and servers characteristics.
For example, some tasks, like machine learning, are CPU intensive while some others, like reduce tasks, are both memory and network intensive.

Data centers are being composed of a very large number of servers with very varied specifications in terms of computing capacities, memory, storage, etc.:

Number of servers	CPUs	Memory
6732	0.50	0.50
3863	0.50	0.25
1001	0.50	0.75
795	1.00	1.00
126	0.25	0.25

Number of servers	CPUs	Memory
52	0.50	0.12
5	0.50	0.03
5	0.50	0.97
3	1.00	0.50
1	0.50	0.06

Configurations of servers in one of Google clusters
(CPUs and Memory are normalized to the maximum server)

Aynchronous hardware upgrades like adding/removing servers, make the resource management in these data centers even more complex.

Many **cluster managers** have been designed to:
orchestrate and harmonize the communication between frameworks and servers, and also improve resource management within data centers.

Apache Mesos is a cluster manager widely used in very large clusters such as Twitter, eBay, Verizon, Airbnb, …
It offers the possibility of developing APIs and supports a large number of frameworks such as Aurora, Chronos, Marathon, etc.

Its *two-layered architecture* ensures an efficient separation between the physical nodes of the data center and the frameworks running on the top level.
Through this architecture, it guarantees *resource sharing in a fine-grained manner*, improving cluster utilization.

Mesos runs on two entities to ensure this separation, on each of the slaves and on the manager (master) node.
The slaves' agents report continuously the state of the hosts to the master:
number of free slots, machine status (up or down).

The process of resource allocation is explained next on an example.

1. A Mesos Slave agent sends periodically to the Master node the state of the host: number of free slots.

2. The Master invokes the *Allocation Policy module* and decides how many resources to offer to the Framework. In the example, the allocation policy module decided to offer all the resources of Slave 2 to Framework 2.

3. Framework 2 processes the received offer; it may accept or refuse it. In the case of acceptance, it sends back to the Master (by means of the *Framework Scheduler module*) the tasks it wants to run, possibly with specifications of placement constraints.

4. The Master sends tasks to the Slave, which will launch them (by means of the slave's *Executor module*).

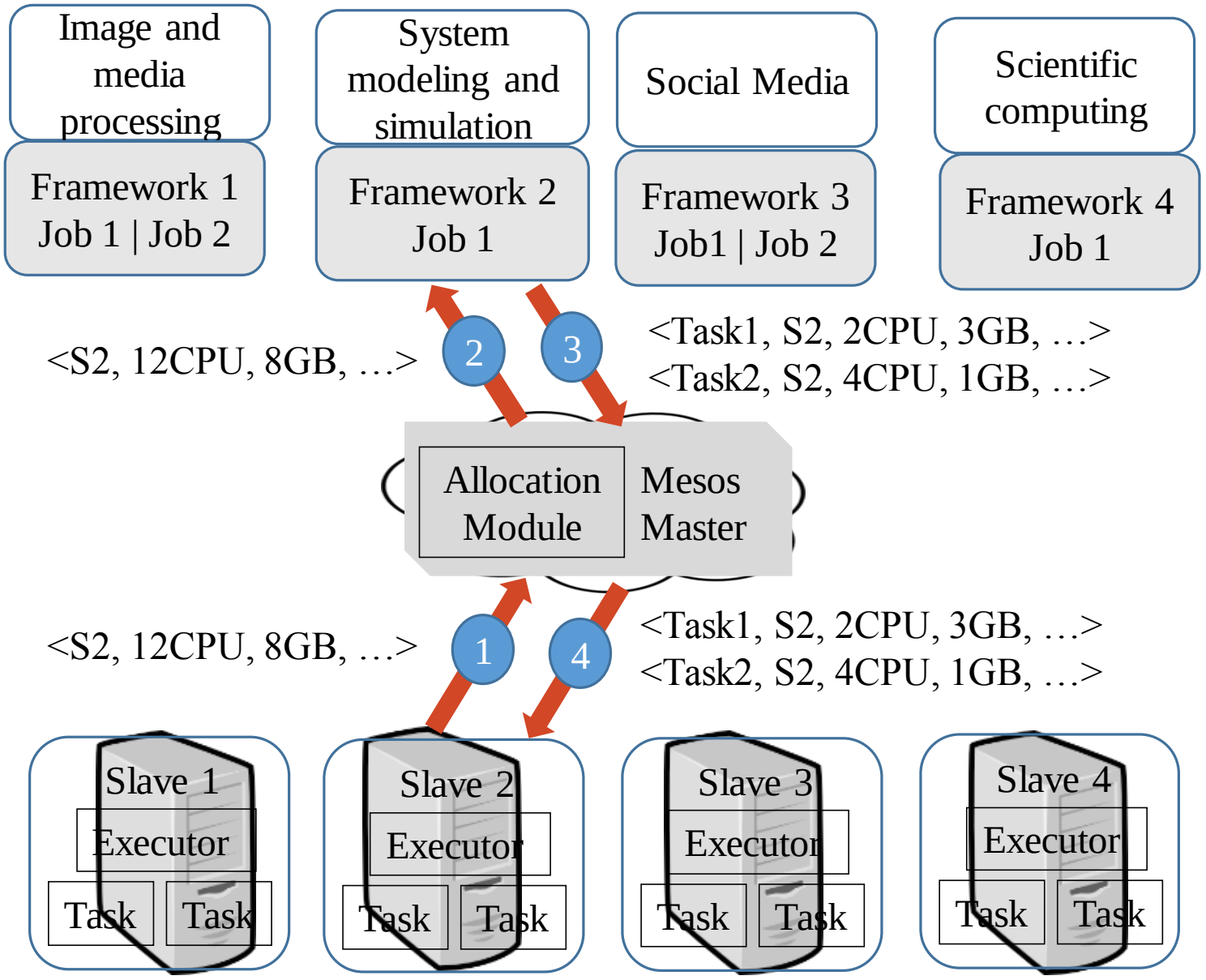
This process will be repeated until all the tasks finish and new resources become free.

The example shows how Mesos performs *two key functionalities* :

- ***resource offer allocation,*** ensured by the Master, and
- ***resource isolation,*** performed by the Slaves.

MESOS

- Popular cluster manager.
- Open Source.
- Developed by Apache.
- Used by at least 50 organizations.



Resource management policies

Sharing a data center's resources between applications is NP-Hard.

Exact algorithms are not used in large-scale data centers mainly because:

- applications are having an unpredictable behavior in terms of arrival rate and resource requests, and
- most of the applications require real-time assignment of resources to their tasks. These systems are called *dynamic real-time systems*.

Because of these constraints, *heuristics* have been implemented to perform *on-line (allocation and) scheduling*.
Among these we consider two scheduling algorithms based on the *user's share:*

- ***Dominant Resource Fairness***
- ***Tetris***

Note: these algorithms support associating weights to the users so that the resource management strategy accounts for user priority.

Dominant Resource Fairness (DRF)

This resource allocation policy aims at guaranteeing an *efficient* assignment of resources to tasks in a multi-resource context while guaranteeing *fairness*.
It is the default resource allocation policy implemented in Mesos.

DRF has been designed to solve the problem raised in algorithms such as Quincy and Hadoop Fair Scheduler, which ignore the heterogeneity in the user's resource demands.

It generalizes *max-min fairness* to *multiple resources*.
Its resource allocation policy is based on the definition of the resource which is utilized the most by the tenant, called the *dominant share*.

DRF has four properties desirable in any *fair resource allocation policy* in the case of heterogeneous demands and multiple resources:

1. users should not be able to improve their allocations by lying about their resource demands *(Strategy-proofness)*,
2. the allocation of a user cannot be improved without decreasing the allocation of another user *(Pareto-efficiency)*,
3. for a user, it should be more beneficial to share the cluster with other users than to use his own partition, this means that the strategy should guarantee at least $1/n$ of the resources for each of the n users *(Sharing Incentive)*, and
4. a user should not prefer the allocation of another user *(Envy-freeness)*.

The principle of DRF is based on computing the dominant share of each user in the system.
The *share* is a quantitative value measuring the *fraction* of resources allocated to the user among the whole system resources.
The *maximum* value of these shares is called *dominant share* and the corresponding resource is the *dominant resource*.

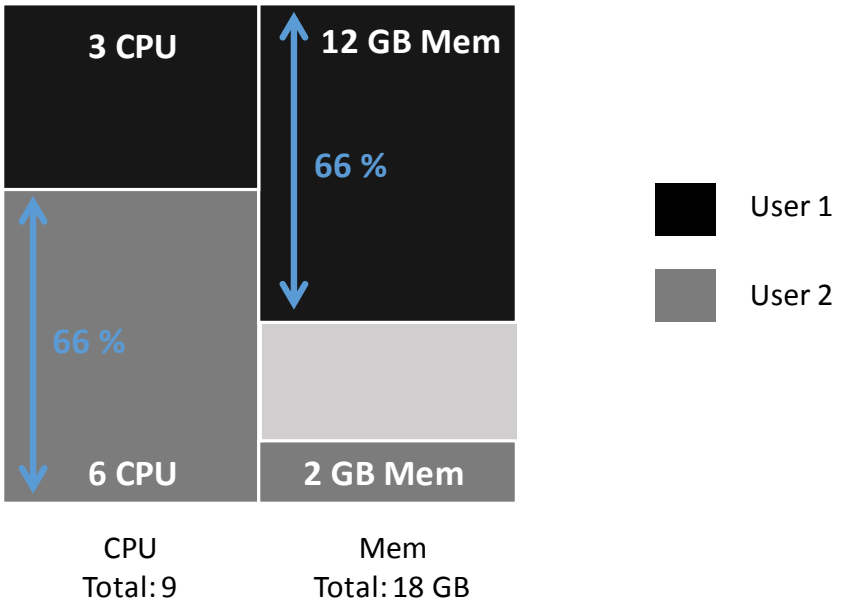
Dominant Resource Fairness aims at equalizing the users shares so that they obtain an equal amount of their dominant resources.

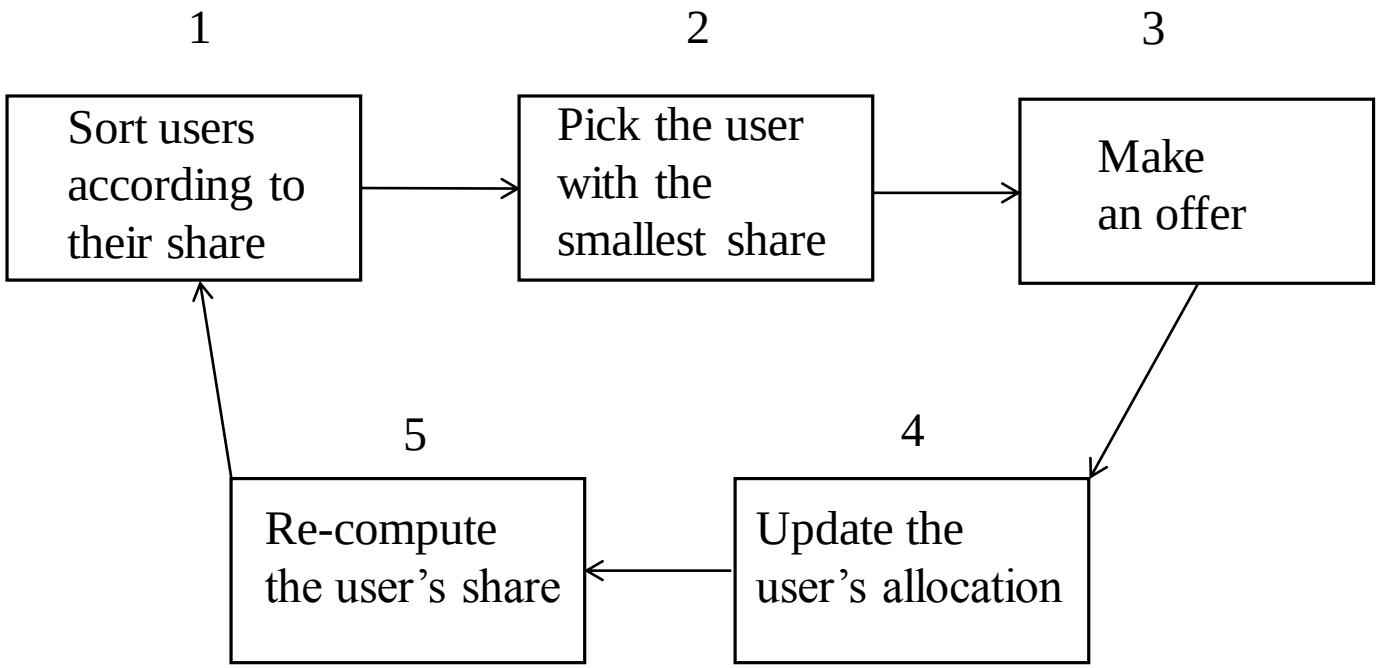
To see how it works, consider a system with 9 CPUs and 18 GB of memory.
Two users A and B are sharing the system, with respective demand vectors:
— < 1 CPU, 4 GB > for user A
— < 3 CPU, 1 GB > for user B
Every task of user A will consume 1/9 CPU and 2/9 Memory,
hence A's dominant resource is Memory.
Every task of user B will consume 1/3 CPU and 1/18 Memory,
hence B's dominant resource is CPU.

DRF will attempt to equalize the users' dominant shares, by allocating the same quantity of Memory to user A as the quantity of CPU allocated to user B.
Applying this algorithm user A gets $< 3CPU; 12GB >$ and user B gets $< 6CPU; 2GB >$
i.e. each user gets 2/3 of its dominant share.

Dominant Resource Fairness (DRF)

- Default allocation policy in Apache Mesos.
- Generalization of max-min fairness to multi-resources.
- Based on the definition of dominant share.
- Keeps an equal share across users.
- **Example:**
  - Total resources: <9CPU,18GB>
  - User1 demand: <1CPU,4GB> 1/9 vs 4/18 dominant resource: Mem
  - User2 demand: <3CPU,1GB> 3/9 vs 1/18 dominant resource: CPU





Main steps of DRF algorithm

Tetris

Most schedulers use slots of fixed size and assign tasks to machines based *only* on *one resource* (for example : dominant resource in the case of DRF).
This results in the fragmentation of resources and a bad exploitation of the data center since some resources will be *over-allocated*.

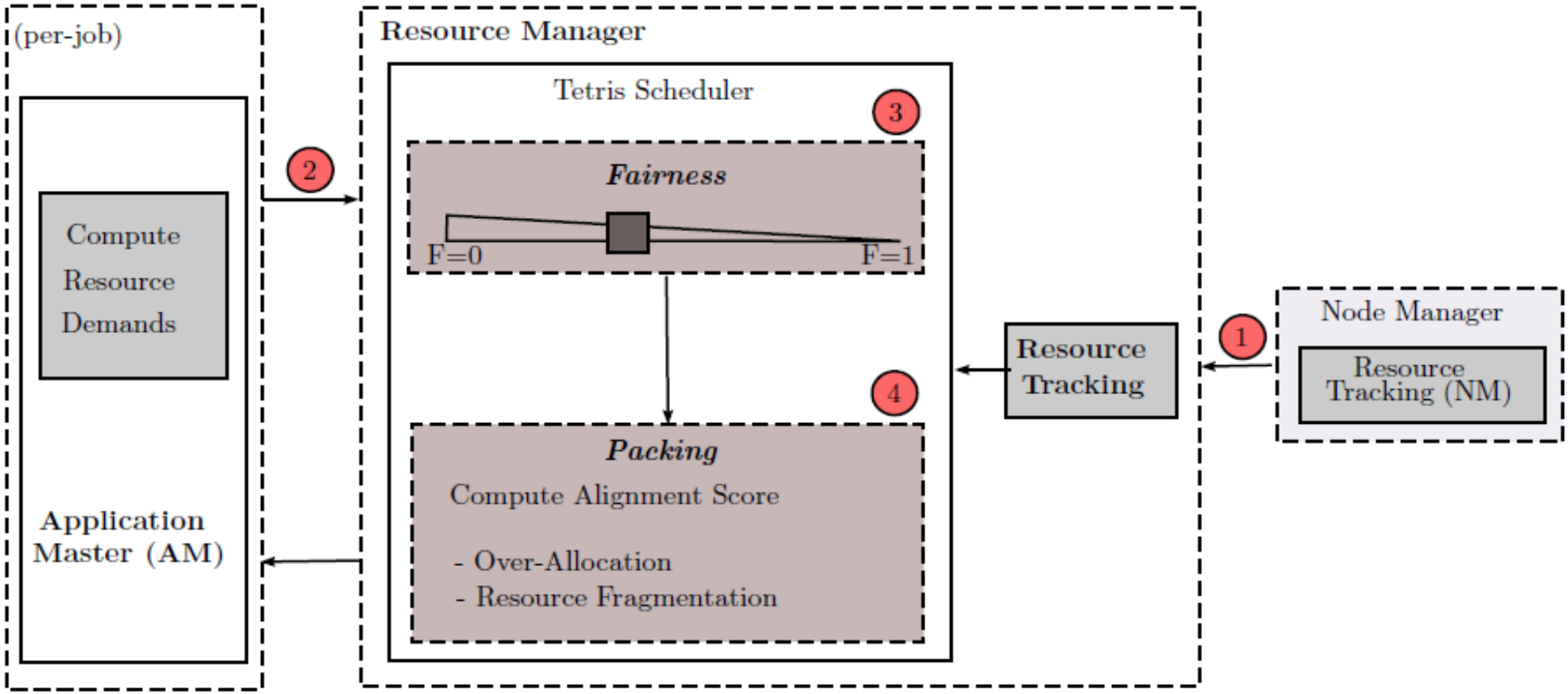
Tetris is a cluster scheduler inspired from the multi-dimensional bin packing problem that tries to improve job scheduling in multi-resource large data centers.

The idea is to consider tasks as balls and machines as bins.
Although *multi-dimensional bin packing* problem is APX-hard, several heuristics exist to give an approximate solution to the problem.
(Note: APX is an abbreviation for *approximable*. APX-hard refers to the set of NP optimization problems that allow polynomial-time approximation algorithms with approximation ratio bounded by a constant.)

Tetris adapts these solutions to the case of varied resource demands size and tasks arrival time. This heuristic for job scheduling based on multi-resource packing reduces makespan and job's average completion time.

The heuristic proposed by Tetris is based on projecting the tasks requirements and the data center's remaining resources in an euclidean space.
Then the couple (task, machine) with the highest dot product is chosen. By using the dot product, the scheduling algorithm will give priority to large tasks and tasks with requirements in resources similar to the available resources.

To explain how Tetris works and to show how it improves Makespan and Data Locality, we present a simplified schema:



1. First, periodically, the Resource Tracking at the node manager sends to the ResourceManager the state of allocation of the data center's nodes: the amount of free/used resources.

2. The Application Master (AM) sends a resource request for its tasks to the resource Manager. Whenever a heartbeat is received from the Node Manager (NM) the Resource Manager runs the following algorithm to match tasks to resources.

3. The Fairness Knob (F) is a parameter of the algorithm, it takes values between 0 and F = 0 implies best efficiency and F = 1 implies perfect fairness.
   The data center administrator is supposed to choose a value that makes a trade-off between efficiency and fairness. Once the value of F is fixed, Tetris starts by ordering the jobs in increasing order.

4. For each job in the selected list of jobs, Tetris computes an alignment score of each of its tasks (*At*). The alignment score is a dot product between the task's requirement vector and the host's available resources vector. This score will be higher for the tasks that match better in the given host.

5. The last step is to pick the task with the highest alignment score $At + \varepsilon Pj$ where ε trades-off the packing efficiency and job completion time. At the end, go to step 3 and repeat the algorithm until all the resources of the host get consumed.

To illustrate the results of applying DRF and Tetris, consider:

- a cluster with 18 cores, 36 GB of memory and 3Gbps network, and
- three jobs A, B and C, each having two phases,

Job	**Tasks**	**Requirements**
A	18 tasks	1 core, 2 GB of Memory
B	6 tasks	3 cores, 1 GB of Memory
C	6 tasks	3 cores, 1 GB of Memory

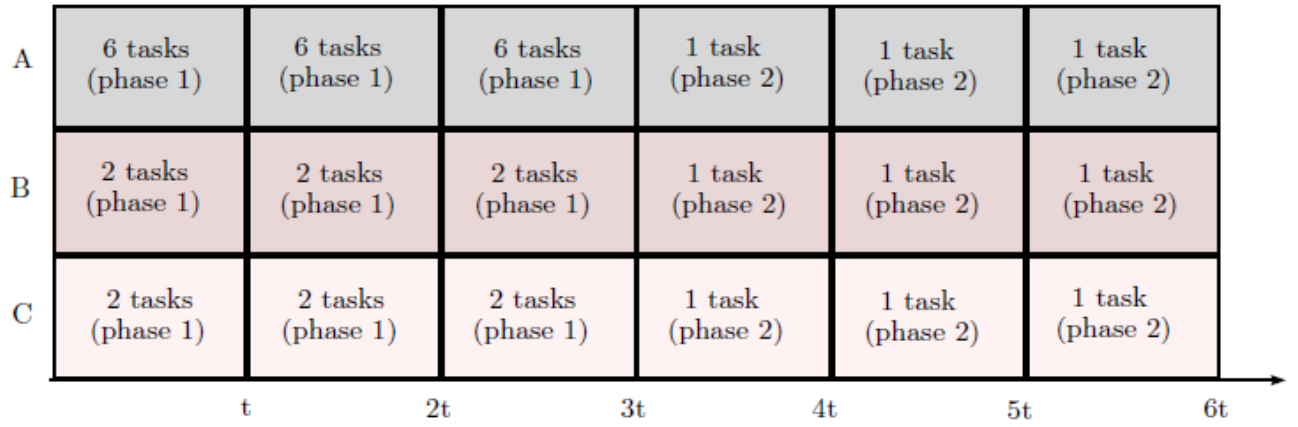
Upon application of the algorithms

- Tetris improves the *job completion time* of all the jobs up to 66 % for job A, 50 % for job B and 33 % for job C.
- It improves *makespan* by 33 % and thus all the jobs finish earlier compared to fair scheduler (DRF).
- It reduces *resource fragmentation*.

All these improvements result from 3 properties of Tetris :

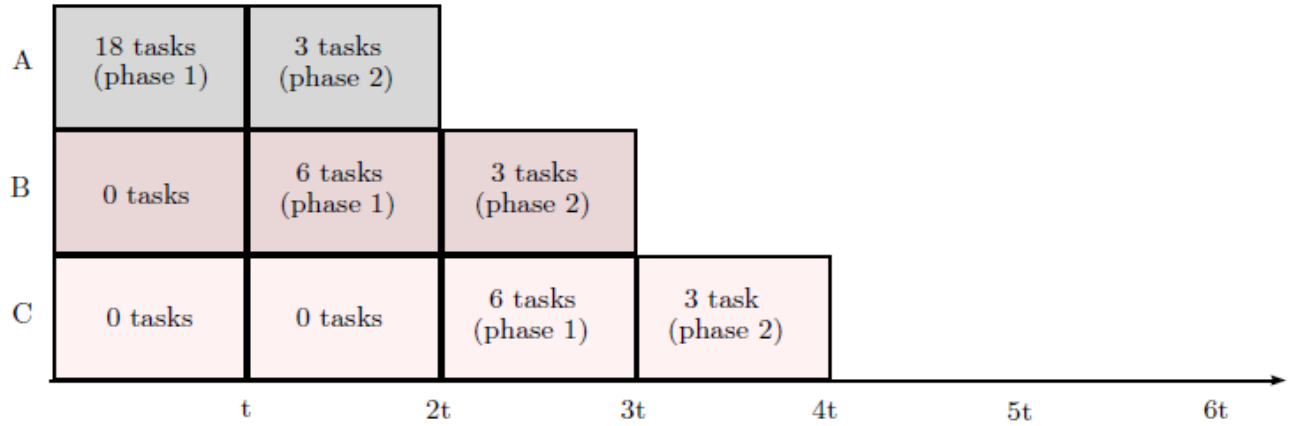
- preferring jobs with fewer remaining tasks,
- avoiding resource fragmentation, and
- computing the best match of tasks to machines using the dot product.

18 cores	18 cores	18 cores	0 cores	0 cores	0 cores
16 GB	16 GB	16 GB	0 GB	0 GB	0 GB
0 Gbps	0 Gbps	0 Gbps	3 Gbps	3 Gbps	3 Gbps



(a) Job schedule under DRF allocation

18 cores	18 cores	18 cores	0 cores		
36 GB	6 GB	6 GB	0 GB		
0 Gbps	3 Gbps	3 Gbps	3 Gbps		



(b) Job schedule with multi-resource Packing (Tetris)