Sort clusters algorithms

September 20, 2020

September 20, 2020

1 Introduction

Something about CAL language and a whole BaltiLSC system and the project basic information ...

2 Data structures

2.1 Cluster-based structures

```
class Machine:
  MEMORY: float # [GB]
  CPUs: List[float] # [GHz]
  GPUs: List[float]
  PRICE: float # [EURO/h]

```

Listing 1: Machine data structure

```
class Resources:
  CPU: List[float] # [GHz]
  GPU: List[float]
  MEMORY: float # [GB]

```

Lets indroduce the structure that represents the class of a machine. It takes into account the generalized resources of the machine across their scope:

```
class MachineClass:
  MIN: Resources
  MAX: Resources

```

Listing 2: Machine data structure

```
class Storage:
  SIZE_RANGE: Tuple[float, float] # [GB]
  TYPE: Enum["SSD", "HDD"]
```

```
    PRICE: float # [EURO/(TB*h)]
    PricePerHour: float

class Cluster:
    ID: int
    MACHINES: Dict[str: Machine]
```

Listing 3: Cluster data structure

2.2 Application-based structures

```
class Data:
    TYPE: Enum['SET_OF_IMAGES', 'CSV_FILE', ...]
```

```
class DataSpec(Data):
    OVERALL_SIZE: float # [GB]
    QUANTITY: int
    LARGEST_PART_SIZE: float # [GB]
```

```
class Module:
    NAME: str
    REQUIRED_RESOURCES: Resources
    INC: float # internal network capacity [Gb/s]
    ENC: float # external network capacity [Gb/s]
```

Listing 4: Module data structure

```
class Application:
    MODULES: Dict[str: Module]
    INPUT_DATA: Data
    OUTPUT_DATA: Data
    CAL_DIAGRAM: str # CAL diagram in .yaml format
    AIFR: func(data_size: float): float # Function that returns
    internal flow ration of the application based on input data
    size*
```

*internal flow ratio will be describe soon

```
class Task:
    APP: Application
    INPUT_DATA: DataSpec
    RESOURCES_RESERVATION_RANGE: Tuple[Resources, Rersources]
```

```
class TaskExecution:
    TASK_ID: int
    CLUSTER_ID: int
    TIME: float # [s]
```

Listing 5: TaskExecution data structure

```
class ModuleExecution:
  MODULE_ID: int
  MACHINE_ID: int
  TIME: float # [s]

```

Listing 6: ModuleExecution data structure

3 Algorithm implemetation

3.1 Input and output data

As a input data for algorith

1. *task_1*: Task - the task we want to execute,
2. *matching_clusters*: List[Cluster] - list of clusters that meet resources reservation range of the *task_1*
3. *time_price_ratio*: float [0,1] - the priority of sorting policy set up by user who is executing the task

As a output data of the algorithm we receive a list of sorted clusters with paired modules for individual machines classes. The example output can look like this (MachineClass, Cluster and Module simplified data structures are shown in listening 2, 3 and 4):

```
[
  (
    c_1: Cluster,
    [
      (m_1: Module, MC_1: MachineClass),
      (m_2: Module, MC_1: MachineClass),
      (m_3: Module, MC_4: MachineClass),
      ...
    ]
  ),
  (
    ...
  ),
  ...
]

```

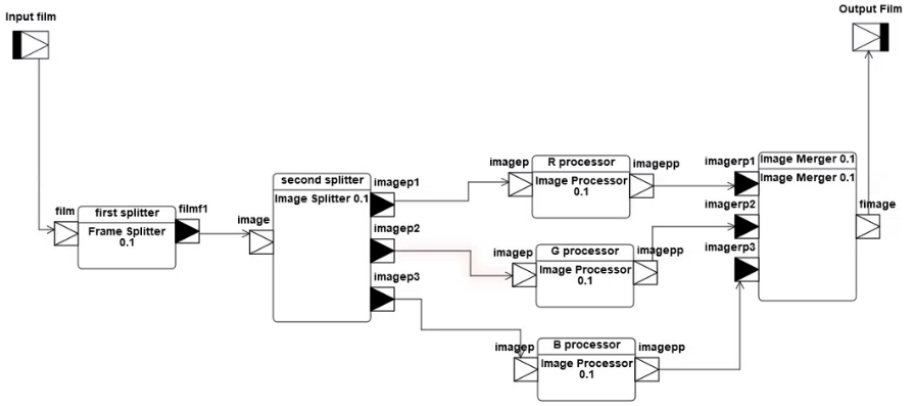
Listing 7: Example output of the sorting algorithm

3.2 Algorithms utils

Let's get through our algorithms with an app as follows:

```
app_1 = Application(
  MODULES = {
    'first splitter': m1,
    'second splittter': m2,
```

Figure 1: *app_1* implementation with CAL language



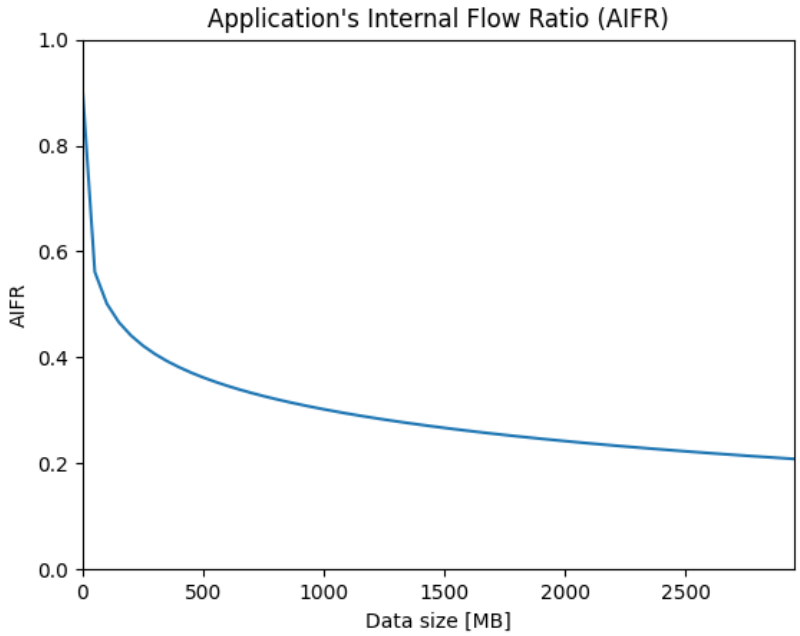

```
    'R processor': m3,
    'G processor': m3,
    'B processor': m3,
    'Image Merger 0.1': m4,
  },
  INPUT_DATA = Data(
    TYPE = 'FILM'
  ),
  OUTPUT_DATA = Data(
    TYPE = 'FILM'
  ),
  CAL_DIAGRAM = '(call diagram content based on application
  implementation)',
  AIFR = func(int dataSize) { ... } # See the figure below
)

```

An example Task structure is listen below:

```
task_1 = Task(
  APP = app_1,
  INPUT_DATA: DataSpec(app_1.INPUT_DATA) (
    OVERALL_SIZE = 1.2
    QUANTITY = 1
    LARGEST_PART_SIZE = 1.2
  ),
  RESOURCES_RESERVATION_RANGE: (
    Resources(
      CPU = [1.8],
      GPU = []
      MEMORY = [2],
    ),
    Rersources(
      CPU = [2.4, 2.4, 2.4, 2.4, 2.4, 2.4],
      GPU = []
      MEMORY = [4],
    )
  )
```

Figure 2: *app_1 AIFR*




```
)
```

Let's assume that we have for a disposition the following clusters:

```
c1: Cluster = Cluster(
  ID: 1,
  MACHINES: {
    '1': Machine = (
      MEMORY = 2
      CPUs = [1.8]
      GPUs = []
      PRICE = [0.12]
    ),
    '2': Machine = (
      MEMORY = 4
      CPUs = [2.2]
      GPUs = []
      PRICE = [0.18]
    )
  },
)
c2: Cluster = Cluster(
  ID: 2,
  MACHINES: {
    '1': Machine = (
      MEMORY = 2
```

```
        CPUs = [1.8]
        GPUs = []
        PRICE = [0.15]
      ),
      '2': Machine = (
        MEMORY = 4
        CPUs = [2.4]
        GPUs = []
        PRICE = [0.25]
      )
    },
  )

```

The helper functions used by sorting algorithms are listed below:

1. getAppModulesLine(app: Application) → List[Module] - return the sequence of app's modules

```
    >>> getAppModulesLine(app_1)
    [m1, m2, m3, m4]

```

3.3 Price-based algorithm

Price-based algorithm takes following assumptions:

1. Each instance of the same *MachineClass* is equal in terms of computation speed, regardless of the cluster it belongs to.
2. Internal data flow time is omitted.
3. The output is a list of sorted clusters without any information about computation times.
4. Algorithm estimates the relative (in relation to other clusters) costs of task execution and do not consider the computation times at all in the final result.

The minimum data set that we need to execute the algorithm consist of the specification for each machine (more specific: a machine resources to assign the appropriate class and the price per hour for using it).

Price-based algorithm steps:

1. getModulesToMachineClassProbability(app: Application, clusters: List[Cluster], rrr: Tuple[Resources, Resources]) → Dict[int: List[Tuple[Module, List[Tuple[MachineClass, float]]]]] - return the list of module to machine class pairs (with probabilities of each matching) for each cluster. It takes into consideration the resource reservation range and the resource requirements for each module. The probability can depends on how fequently the modules are assigned to each MachineClass (the extended version of the algorithm using historical data). In this simple example we assume uniform distribution of probabilities and that each module can be executed on any class of a machine.

```
>>> getModulesToMachineClassProbability(app_1, [c_1, c_2],
 (resources_min, resources_max))
{
  1: [
    (m1, [
        (mc_1, 0.5),
        (mc_2, 0.5),
      ]
    ),
    ...
    (m4, [
        (mc_1, 0.5),
        (mc_2, 0.5),
      ]
    ),
  ]
  2: [
    (m1, [
        (mc_1, 0.5),
        (mc_2, 0.5),
      ]
    ),
    ...
    (m4, [
        (mc_1, 0.5),
        (mc_2, 0.5),
      ]
    ),
  ]
}

```

2. In the list of modules to machine class probabilities for each cluster we change each machine class instance to it price per hour and receive:

```
{
  1:[
    (m1, [
        (0.12, 0.5),
        (0.18, 0.5),
      ]
    ),
    ...
    (m4, [
        (0.12, 0.5),
        (0.18, 0.5),
      ]
    ),
  ]
  2: [
    (m1, [
        (0.15, 0.5),
        (0.25, 0.5),
      ]
    ),
    ...
    (m4, [
```

```
            (0.15, 0.5),
            (0.25, 0.5),
          ]
    ],
  }

```

3. Next, we make internal multiplication of elements of List for each module:

```
  {
    1:[
      (m1, 0.15),
      ...
      (m4, 0.15),
    ]
    2: [
      (m1, 0.2),
      ...
      (m4, 0.2),
    ]
  }

```

4. Finally, we can sum the prices for each module execution with appropriate weights. Weigths are calculated based on time execution ratios between modules and the size of the input data (in this simple example we assume equal execution times among all of the modules):

```
  >>> getModulesTimeExecutionRatios(task_1: Task) -> List[
  Tuple[Module, ratio]]
  [(m1, 0.25), (m2, 0.25), (m3, 0.25), (m4, 0.25)]

```

The result of price-based algorithm based on our example:

```
  [(1, 0.15), (2, 0.2)]

```

The algorithm accuracy could be increased by better methods of estimating time execution ratios. This methods will be described in the following parts.

3.4 Test-execution-based algorithm

No possesion for the historical data is required by test-execution-based algorithm. It only uses data from the set of testing tasks executions. Lets say that we have 3 clusters we want to sort:

```
clusters_list: List[Cluster] = [c1, c2, c3]

```

Listing 8: List of available clusters

On the each of the clusters we run the same set of testing tasks consist of a single JobBatch.

```
testing_tasks: List[Task] = [t1, t2, t3]

```

Listing 9: Set of testing tasks

We end up with the testing tasks performances summaries:

```
testing_tasks_performances: List[Tuple(Cluster, Task, float,
  float)] = [
  (c1, t1, 0.21, 0.11),
  (c1, t2, 0.44, 0.31),
  (c1, t3, 0.24, 0.13),
  (c2, t1, 0.66, 0.07),
  (c2, t2, 0.84, 0.04),
  (c2, t3, 0.54, 0.09),
  (c3, t1, 0.39, 0.21),
  (c3, t2, 0.57, 0.24),
  (c3, t3, 0.33, 0.12),
]

```

Listing 10: Testing tasks performances summaries

The two ending floats are respectively the execution time (in hours) and the execution price (in EUR). Having such an information we can count price and time clusters coefficients. For each testing task we search the minimal execution time and execution price. Then for each cluster we divide each time and price by the minimal values corresponding to appropriate tasks. We end up with the following coefficients:

```
testing_tasks_coefficients: List[Tuple(Cluster, Task, float,
  float)] = [
  (c1, t1, 1.0, 1.57),
  (c1, t2, 1.0, 7.75),
  (c1, t3, 1.26, 1.00),
  (c2, t1, 3.14, 1.0),
  (c2, t2, 1.91, 1.0),
  (c2, t3, 2.84, 1.44),
  (c3, t1, 1.86, 3.0),
  (c3, t2, 1.3, 6.0),
  (c3, t3, 1.0, 1.33),
]

```

Listing 11: Testing tasks performances summaries

Next, we group the table above by cluster with sum as a aggregation function for the price and time coefficients.

```
cluster_coefficients: List[Tuple(Cluster float, float)] = [
  (c1, 3.26, 10.32),
  (c2, 7.89, 3.44),
  (c3, 4.16, 10.33),
]

```

Listing 12: Cluster coefficients

Finally, using the user typed time-price priority (a coefficient in range [0, 1], let's assume it's 0.5 in our simple example) we can count the result:

```
  [
    (c1, 3.26*0.5 + 10.32*0.5),
    (c2, 7.89*0.5 + 3.44*0.5),
    (c3, 4.16*0.5 + 10.33*0.5),
  ]
  final_result = [
    (c2, 5.67),
    (c1, 6.79),
    (c3, 7.25),
  ]

```

Listing 13: any

3.5 Advance algorithm

The algorithm described above is based on execution of testing tasks and the historical data of task executions. To make it work we need to establish the set of well diversified testing applications and modules. The sorting is based on time ratios for the execution of test tasks, so the algorithms used by modules from test tasks should have linear computational complexity to make sorting most reliable.

Every new cluster wanting to join the fun should run each application with a few different input data sets. This data sets should be properly establish as well. Every new machine appering as availible on the cluster should perform computations of all of the modules from the testing task set (if the machine conforms resources requirements of modules of course). Futhermore, we introduce the concept of machine class to generalize the calculation of machine performance.

After performing tests described above we will end up with testing executions data required for sorting algorithm implementation. In listenings 5 and 6 you can find simplified format of the testing tasks executions data.

Sorting has the following features:

1. Each cluster is tested for it internal data time flow beetwen nodes and the sorting is using this times,

2. Each machine on cluster is tested for it performance time and the sorting is using this times,

3. User before starting the task execution can provide time_price_ratio setting a priority on a more time or financial basis,

4. The algorithm not only indicate the most favorable cluster, but also pair the task modules to the appropriate machine classes.

Individual steps of the sorting algorithm:

1. getTimeLines(*task_1*: Task) → List[List[Tuple[Module, int]]] - this will return a list of sequence of modules that will be carried out in succession with the number representing the parallelism of the module. It will return the list of all possible time lines based on the configuration of the application and the *task_1.RESOURCES_RESERVATION_RANGE*.

```
>>> time_lines: List[Module] = getTimeLines(task_1)
>>> print(time_line)
[
  [(m1, 1), (m2, 1), (m3, 1), (m4, 1)],
  ...,
  [(m1, 1), (m2, 4), (m3, 12), (m4, 1)]
]

```

2. getDetailedTimeLines(*time_lines*: List[List[Tuple[Module, int]]], *task_1*: Task, *matching_clusters*: List[Cluster]) → Dict[Tuple[Cluster, str]: List[Tuple[Module, int, MachineClass]]] - return collection of modules list paired with machines (based on *matching_clusters* list and *task_1.RESOURCES_RESERVATION_RANGE*)

```
>>> detailed_time_lines: Dict[str: List[Tuple[Module,
Machine]]] = getDetailedTimeLines(time_line, task_1, [c1,
c2])
>>> print(detailed_time_line)
{
  (c1, 'dl_0'): [(m1, 1, m_c_1), (m2, 1, m_c_1), (m3, 1,
m_c_1), (m4, 1, m_c_1)],
  ...,
  (c2, 'dl_0'): [(m1, 1, m_c_1), (m2, c2.MACHINES['1']), (
m3, 1, m_c_1), (m4, 1, m_c_1)],
  ...
  (c2, 'dl_15'): [(m1, 1, m_c_2), (m2, 1, m_c_2), (m3, 1,
m_c_2), (m4, 1, m_c_2)],
  ...
  (c2, 'dl_767'): [(m1, 1, m_c_2), (m2, 4, m_c_2), (m3,
12, m_c_2), (m4, 1, m_c_2)],
}

```

3. getInternalFlowCoefficient(*task_execution_1*: TaskExecution, c1: Cluster) → float - returns a time coefficient of internal data flow between modules for a given TaskExecution performed on cluster. To be more specific, it returns overall time of task execution minus sum of the maximum executions times of each modules from the task time line. This function using following formula:

$$coeff = \frac{internal\_flow\_time}{task\_execution\_time} \tag{1}$$

In our example (*app_1*, modules parallization respectively equals 1, 3, 4, 1) shown in Figure 3 the coefficient will eqaul:

$$coeff = \frac{T - t1 - t2 - t3 - t4}{T} \tag{2}$$

So as a result we get the internal flow coefficient (IFC) in range [0, 1] the

Figure 3: *app_1* timeline

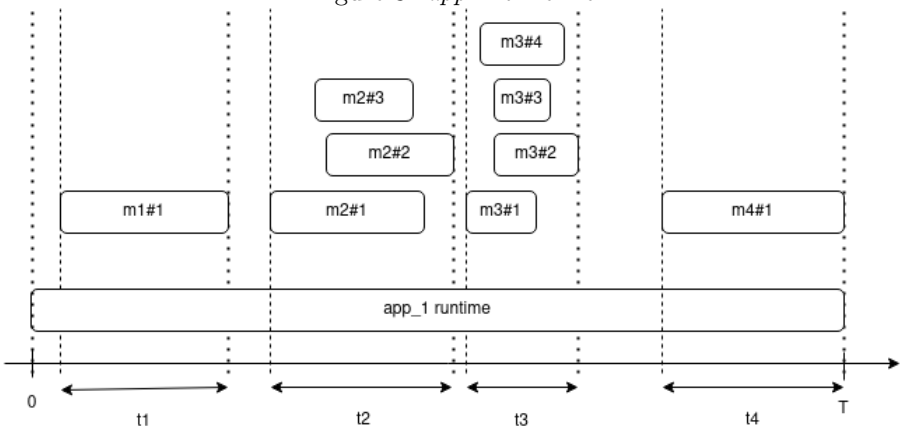
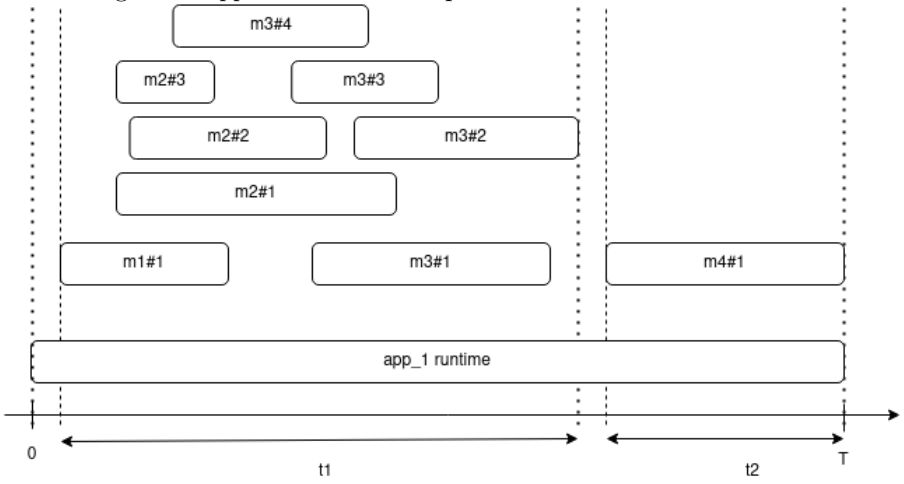


Figure 4: *app_1* timeline with parallelization beetwen modules



value of which reflects the ratio of internal data flow between modules time and the task execution time. If there is no need for modules to be executed in sequence (one after another), it is possible that later modules will be executed in parallel with the earlier ones. This is shown in Figure 4. In such a case the IFC will be significantly lower. Even though, it is possible to count the IFC for such a cases, the algorithm implify the modules are executed one after another.

4. getInternalFlowsRatios(*test_tasks_executions*: SetTaskExecution, *task_1*:

Task, clusters: Set[Cluster]) → Dict[Cluster: float] - count the average internalFlowCoefficient for each cluster from clusters set and returns the collection of ratios of received coefficients. The final ratios are the result of multiplying normalized internal flow coefficients by the task internal flow coefficient.

```
>>> internal_flows_ratios = getInternalFlowsRatios(
test_tasks_executions, [c1, c2])
>>> print(internal_flows_ratios)
{
  c1: 1.19,
  c2: 1.0
}

```

5. getTestingModulesExecutionTimesRatios(test_tasks_executions: List[TaskExecution], detailes_time_lines: Dict[Tuple[Cluster, str]: List[Tuple[Module, int, MachineClass]]]) → Dict[str: List[float]] - we take first Tuple[Module, int, MachineClass] from each element of detailes_time_lines. Then we create an unique set of machine classes with number of parallization divided (with rounding up) by availible parallization based on machine class resources range (so if we have 4 availible CPUs for a machine class and the parallization coefficient for the modul under consideration is 10, then have to multiply module execution time by roundUp(10/4) = 3). In our example it will be just:

```
machines_1 = { (m_c_1, 1), ..., (m_c_1, 1), ... , (m_c_1,
1), ... , (m_c_1, 1) }

```

So for the first Tuple[Module, int, MachineClass] the rations will equal 1. Based on test_tasks_executions we can extract modules executions common to set of machines described above. Lets say it will be a following set:

```
m_1_mod_execs: Set[TaskExecution] = {t1m1, ..., t1m4, ...,
 t5m2, ..., t6m1}

```

Having information about the execution times of modules on machine classes, for each module we count execution time ratio for each machine class (taking into consideration the number of parallization for each module):

$$t1m1\_c1m1 = \frac{t1m1.TIME/\#\{module\_parallization\}}{[\text{t1m.TIME for t1m in m\_1\_mod\_execs}].sum()} \quad (3)$$

So for each machine class we will receive a set of modules executions times ratios:

```
c1m1: Set[float] = {t1m1_c1_m_c_1, ..., t1m4_c1_m_c_1,
..., t5m2_c1_m_c_1, ..., t6m1_c1_m_c_1}

```

Then we count the average ratio of testing modules executions for each class machine:

```
    c1m1_ratio: float = {t1m1_c1_m_c_1, ..., t1m4_c1_m_c_1,
    ..., t5m2_c1_m_c_1, ..., t6m1_c1_m_c_1}.average()

```

By carrying out the above calculations for each column from detailed_time_lines we receive modules executions times ratios:

```
    >>> times_ratios = getModulesExecutionTimesRatios(
    test_tasks_executions, [c1, c2])
    >>> print(times_ratios)
    {
      (c1, 'dl_0'): [1.41, 1.0, 1.02, 1.20],
      ...,
      (c2, 'dl_0'): [1.32, 1.55, 1.28, 2.04],
      ...,
      (c2, 'dl_15'): [1.32, 1.55, 1.28, 2.04],
      ...,
      (c2, 'dl_767'): [1.0, 1.08, 1.1, 1.0],
    }

```

6. getPricingMachinesRatios(detailes_time_lines: Dict[str: List[Tuple[Module, Machine]]], [c1, c2]) → Dict[str: List[float]] - for each column from detailed_time_lines we count the ratios of machines pricing:

```
    >>> pricing_ratios = getPricingMachinesRatios(
    test_tasks_executions, detailes_time_lines, [c1, c2])
    >>> print(pricing_ratios)
    {
      (c1, 'dl_0'): [1.0, 1.0, 1.0, 1.0],
      ...,
      (c2, 'dl_0'): [1.0, 1.11, 1.08, 1.00],
      ...,
      (c2, 'dl_15'): [1.25, 1.25, 1.25, 1.25],
      ...,
      (c2, 'dl_767'): [2.08, 2.08, 2.08, 2.08],
    }

```

7. getTimePricingRatios(detailes_time_lines: Dict[str: List[Tuple[Module, Machine]]], [c1, c2]) → Dict[str: float] - this function return the ending result of the algorithm as a time-pricing ratios which are composed of two parts:
   (a) Time part, calculated using the formula below:

$$t = detailed\_time\_line.sum() * (1 + app\_1.AIFR(task.input\_data) * cluster\_IFR) \tag{4}$$

   (b) Price part:

$$p = (detailed\_time\_line * pricing\_rations).sum() \tag{5}$$

We normalize this part through the all detailes_lines (is it necessary?) and multiply the parts by the user price-ratio parameter respectively:

$$tpr = time\_price\_ration * t + (1 - time\_price\_ratio) * p \tag{6}$$

```
>>> time_pricing_ratios = getTimePricingRatios(
  internal_flows_ratios,
  times_ratios,
  pricing_ratios,
  time_price_ratio
)
>>> print(pricing_ratios)
{
  (c1, 'dl_0'): 1.37,
  ...,
  (c2, 'dl_0'): 1.39,
  ...,
  (c2, 'dl_15'): 1.21,
  ...,
  (c2, 'dl_767'): 1.0,
}
```

It needs to be highlighted that we sort clusters for a JobBatch. JobBatch is a part of a Task and if we consider a Task with a few JobBatches we have to include data transfer time beetwen cluster in the sorting algorithm. It will be realized through introducing the data tranfer time coefficient (DTTC). For instance, if a previous JobBatch was executed at Cluster 'A', then actual JobBatch will have the DTTC equal to one for Cluster 'A' and eqaul to more than one to Cluster 'B'. Time-pricing ratios will be multiply by the corresponding DTTC and thus we will get the final time-pricing ratios. The smaller time_pricing ratio a time line have the better it suits the user expectations.

4 Algorithms tests

5 Conclusions