Scheduling (Part II)

Davide La Rosa

May 12, 2010

6 Grid scheduling

In this situation we can have normal machines, clusters, heterogeneous clusters, each one with its internal scheduler and a grid user that is interacting with all the grid resources from a single point. We will have several users of the grid that are interacting from different point with the same grid middleware. The grid middleware is placed between those who stay above (the users that want to execute their jobs) and those below (local schedulers of the various resources). There are several levels of scheduling, that means there are several levels in which we have to take scheduling decisions.

We have schedulers at resource level: if we connect our single computer to the network, it will execute its job using the default scheduler (for instance linux, real-time, torque). For instance, Globus provides a common independent interface to all the scheduling mechanisms; indeed to start a process on linux we have to execute a fork() and an exec() while to start a process on a cluster we have to make a little script that has different syntax depending on the scheduler we are using.

Then we have the enterprise level schedulers: they are designed to communicate with local schedulers and they manage resources at an administrative domain in which the security constraints are more relaxed.

The last are the grid level schedulers: also called brokers, meta-brokers, super-schedulers, their job is to find the way through all the schedulers connected to them, towards the resources needed for the job execution. These schedulers are requested to take real time decisions because the grid is continuously mutating (resources appear and disappear, dynamic security, ...).

The following are the 11 macro-actions that every grid scheduler performs:

Resource discovery

1. **Authorization filtering:** handle with the security mechanisms (find the resources to which the user is authorized to access).
2. **Application definition:** take the description of the job that the user has provided and extract the parameters on which choose the resources.
3. **Minimum requirement filtering:** take into account the job requirements and a possible user defined objective function.

System selection

4. **Information gathering:** interrogate the MDS (updated almost in real-time on the resources status) to find and get information on all the available resources within a given timeout in order to execute the job with the provided constraints.

5. **System selection:** check if given all the available resources, exists an admissible scheduling for the job using one or more algorithms (we have a hierarchical scheduling, so there are a lot of complications). If more than one solution is found, the best is chosen based on the criteria that the user has provided.

Job execution

6. **Advance reservation:** reserve time slots on each machine that will be used.

7. **Job submission:** send the job on the local resource.

8. **Preparation tasks:** for instance copy the executables on the destination resources creating, if necessary, a sandbox directory to prevent interactions with other local users, transfer the input files that maybe encrypted, ...

9. **Monitoring progress:** the broker or something acting on its behalf should monitor the job execution to verify if there is any problem, in case reschedule some of the components or if it is necessary, comply with precedence rules (for instance expressed with a DAG).

10. **Job completion:** the results have to be taken back to the local machine.

11. **Clean-up tasks:** remove any traces of the job from the local machines to bring them back to their correct working.

For a scheduler these steps are always necessary, even if sometimes it is possible to skip some of them because they are performed by the local resources. Now, we will discuss more in detail how the scheduler should behave.

6.1 Scheduling concerns

First of all the scheduler has to use a language with which the users can express what they want to do without explicitly indicate the resources location (run an application with a given execution constraints, minimizing a given objective function). Then the scheduler will look for the resources that meet the requirements and that satisfy the owner policies (for instance the maximum number of grid job allowed).

In the resource selection phase, the scheduler has to cope with the fact that some MDS information provided by the users could be missing, inaccurate or completely wrong. Moreover, we have to keep in mind that the resource administrator is the only one that has the complete control of the resource, hence he may ignore the scheduler requests. Another aspect to consider, difficult to foresee, is the job's queue waiting time based on the resource state. Usually

prediction heuristics (based on the history logs and future resource load, availability, ...) are used.

There are a lot of things that, when a grid scheduling algorithm is designed, must be taken into account, usually they are called client-side criteria and server-side criteria. When the scheduler has found several resources that satisfy the user execution requirements, it has to choose the best among them. To do this, it uses both the user selection criteria and even the resource provider selection criteria (for instance limits on job size or number, data affinity, ...). The handling of all of these criteria becomes quickly complex and difficult to manage, hence, usually the criteria being used are those called best-fit: according to some heuristic, choose the resources which adapt at most to execute the user jobs (very often it means those that minimize the time needed to complete the job). Complex negotiation mechanisms are required when the user and the resource administrator criteria don't go in the same direction.

Another problem that didn't arise up to now, neither on the single machine (because it's impossible) nor in the clusters (because they are managed in space sharing) is the co-allocation problem. This problem turns out when a job has to be carried out using several resources at the same time (for instance a job requesting 64 CPUs on a grid in which all the resources have 16 CPUs). In this case the application has to be spread on several machines in the grid, each of them with different local scheduler and eventually in different administrative domains with different job management policies. A lot of supplementary information and low-level mechanisms are actually necessary in order to face this issue.

To take all the decisions, the grid scheduler must receive very specific and detailed information from the local schedulers, substantially it has to access to the local scheduling mechanisms even if often they are not accessible or are not reliable. All these mentioned problems come out when we want to perform an high-level scheduling.

To find a solution within a short time, it has been tried to relax all the previous problems and to get a quite good solution. Unfortunately, all the solutions obtained even from the simplest problem, were completely random and it was not possible to demonstrate if they were good or not. We can simply compare them in the average behaviour. The most studied application types for the computational grids are:

- **Bag of tasks:** bag of independent applications more or less parallel (even all sequential).
- **Workflows:** sequence of activities regulated by precedence constraints, usually coded with DAG (Directed Acyclic Graph).

Since the problem is NP-complete, we can apply two methods: a meta-heuristic (an heuristic that is valid for an heaps of problems) or an heuristic (algorithms that work only for particular cases). In the following we will see specific heuristics for bags of tasks and workflows with heterogeneous resources.

6.2 Bag of tasks

Min-Min heuristic

In this heuristic, each task, for each resources at its disposal, has an associated execution time which is somehow provided (usually with an execution times matrix). For each task, this heuristic determines the minimum completion time over all the machines. Among all these minimums, it looks for the minimum completion time that means: "among all the tasks on these resources, which of them will be completed first if we gave to it the best resource?". The task which answers to the question will be scheduled on the best available resource and the procedure starts again until all the tasks have been scheduled. Clearly, each time a task is assigned to a resource, the completion times of the remaining tasks on that resource increase.

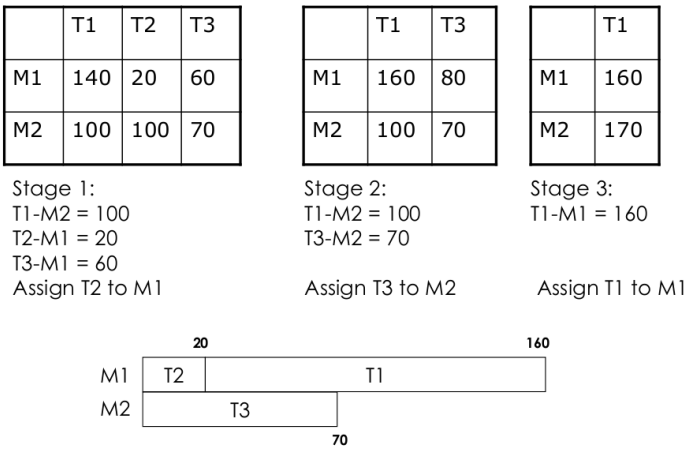

Figure 1. Example of Min-Min heuristic

The performance of this algorithm strictly depends on the foreseen completion times of the tasks. The best way to obtain reliable completion times is to observe the behaviour of the tasks on the resources over a past given amount of time. Then we can calculate the mean (or even minimum, maximum, mode) completion times using the previous information. In any case the quality of the solutions is not guaranteed even if usually this algorithm finds quite good ones.

Max-Min heuristic

The rationale on which this algorithm is based is to assign the longest jobs to the fastest machines and to keep the slower ones for the smaller jobs. In this case, each time we assign a task to a machine, the completion time of the tasks not yet assigned on the same machine will grow very quickly.

With the same data of the previous example, in this case the total completion time is improved, but this doesn't mean that the algorithm is better, it simply depends on the numbers provided.

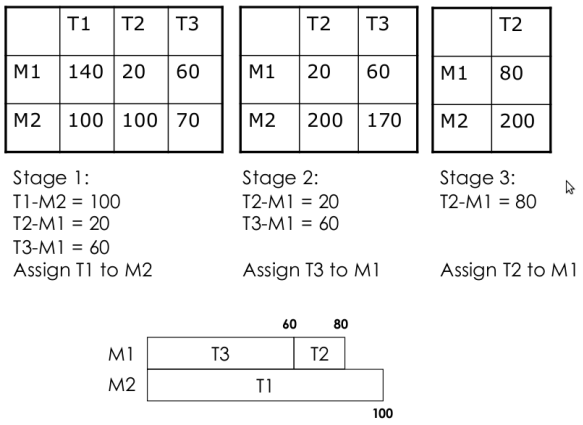

Figure 2. Example of Max-Min heuristic

Sufferage heuristic

In this heuristic the concept is the following: for each task we determine the difference of the completion times (called task suffering) between the first and the second fastest machines. In other words it is the penalty that has to be paid if the task is scheduled on the second best machine instead of the first one. In each round we schedule the task which has the higher suffering, that is the one which would have been penalized at most if it had been assigned to the non optimal resource.

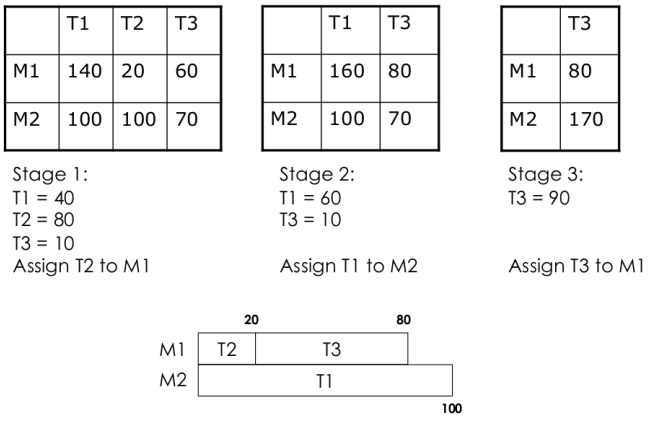

Figure 3. Example of Sufferage heuristic

In the example is clear that, in the first step, T2 undergoes a big penalization if it is scheduled on the wrong machine, while for T3 there is almost no difference. It is important to keep in mind that all the machines are completely uncorrelated, this means there is no relations between the completion times.

6.3 Workflows

With workflows, the tasks are subject to dependencies among them, and since in practice the number of nodes can reach several thousands, find a good scheduling becomes an extremely difficult job. There exist optimal solutions for very particular instances of problems like chains or trees made by one node and several leaves, but in the general case it is impossible.

To obtain a solution we can use meta-heuristics like genetic algorithms, simulated annealing, local search techniques or heuristics that we will see in the following.

Short-Sighted heuristic

This is the simplest algorithm, it is based on the concept of ready task. A task is marked as ready when all the tasks from which it depends have been completed. All the ready tasks are inserted in a queue without ordering (random), the first task is extracted and scheduled on the best available resource. When a task is completed, it is possible that other tasks became ready and then they will be put in the queue. The algorithm continues in this way until the ready tasks queue is empty or the available resources are exhausted. This approach is not very smart, so new algorithms have been studied.

List Scheduling heuristic

An improvement of the previous algorithm can be to assign a priority to each task in such a way that every time a new job has to be scheduled, we first order the ready list and then we extract the topmost task (which is the best). To choose the resource on which to execute the task we can apply one of the algorithms seen before, Min-Min for instance, but we will stop at the first step.

Level by Level heuristic

This scheduler partitions the tasks dependencies graph into levels, so that all the tasks within a level are independent. In other words it should be possible

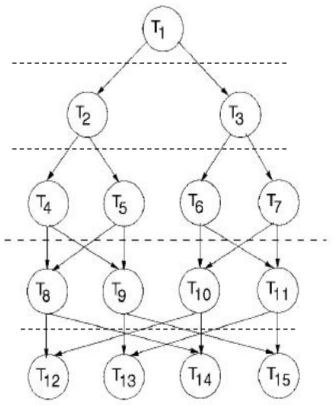

Figure 4. Example of levels partitioning

to replace all the nodes inside a level with one single node and with a single edge going down to the next level. All the tasks belonging to a level will be scheduled using again one of the already seen algorithms for bags of tasks.

This is similar to the List scheduling except that the algorithm runs only between one level and another (the dotted line in the figure 4). Moreover, as long as all the tasks of a level have not been executed, no other jobs on the levels below can be scheduled.

Clustering heuristic

This heuristic tries to look at the workflow as a set of more or less independent sub-workflows, called clusters, with their own relative dependencies. Each cluster can be further scheduled using List, Level by Level or Clustering heuristics.

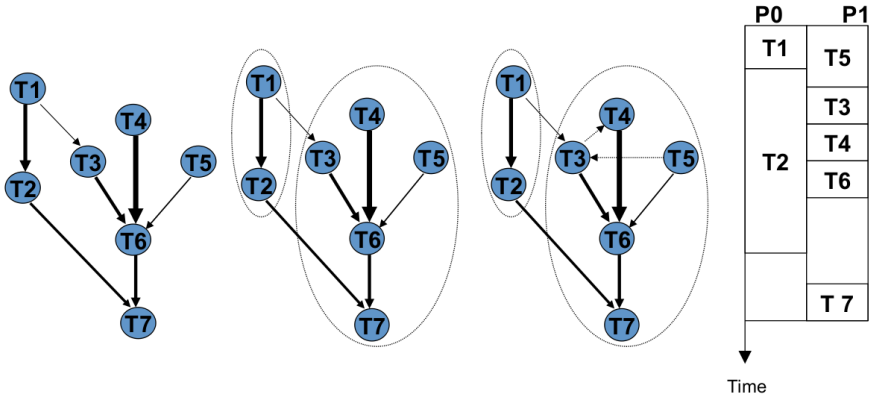

Figure 5. Example of Clustering heuristic

This algorithm is suitable with workflows that have data transfers among the nodes since usually all the tasks belonging to the same cluster are scheduled on the same machine. In figure 5 the arrows have different thicknesses depending on the amount of data flowing between the couples of nodes. In practice, the data transfers inside a cluster don't involve any overload with regard to the data transmission time between machines (provided that each cluster is executed on the same resource). The scheduler can try to group together the tasks in order to minimize the data flow between different clusters.

Heterogeneous Earliest Finish Time (HEFT) heuristic

This algorithm is very used when the workflows have communication costs. We will see the formal definition of this model in detail.

We have n resources indicated as r_j, m tasks indicated as t_i and the weight of the edges indicated as e_{ij}.
Lets indicate with:

$$time(t_i, r_j) = \omega_{ij}$$

the time needed to execute the task t_i on resource r_j and with:

$$time(e_{ij}, r_p, r_q) = c_{ijpq}$$

the time needed to transfer the data e_{ij} from task t_i on resource r_p to task t_j on resource r_q. Now, for each task t_i, we compute the average execution time:

$$\overline{\omega}_i = \frac{1}{n} \sum_{k=1}^{n} \omega_{ik}$$

For each edge e_{ij}, we compute the average communication cost:

$$\overline{c}_{ij} = \frac{1}{n^2} \sum_{p=1}^{n} \sum_{q=1}^{n} c_{ijpq}$$

At this point we rank every node of the workflow starting from the end nodes (those which have no output edges):

$$\forall \text{ end task } t_i \qquad rank(t_i) = \overline{\omega}_i$$

and then moving to the others (calculated recursively):

$$\forall \text{ not end task } t_i \qquad rank(t_i) = \overline{\omega}_i + \max_{t_j \in succ(t_i)} (\overline{c}_{ij} + rank(t_j))$$

where $succ(t_i)$ is the set of all the nodes for which t_i is a predecessor in the workflow.

We start from the lowest level and since each nodes in this level have no data transfers, the ranks are only the average execution time. Then we move back one step and the ranks of the nodes are calculated as their execution time plus the maximum, among all the successors, of the average communication costs plus the rank of the latter. We continue moving back in the workflow until we reach the top tasks. When the rank computation is completed, the nodes are sorted by rank in a decreasing order and they are scheduled accordingly starting from the first. Given a task, to select the machine on which it will be executed, we simply take the available one that gives the minimum execution time. This algorithm is quite complex but generally it performs very well compared to the other heuristics.

Economic heuristic

All the heuristics we have seen up to now, are best-effort heuristics, it means that since they have no budget constraints, they are not trying to optimize the user objective function (for instance money costs) but only the completion time. The economic heuristic aims to reduce the costs of the job executions. Two kinds of objectives are involved: user's ones and system owner's ones. Obviously, they don't go in the same direction since the user wants to reduce the money invested while the system owner wants to maximize the revenue.

In this case the scheduler acts as a broker, given a job it will ask for offers to various resource providers, it will collect all the answers and it will choose the best among them, which is the one that optimizes the user objective function. The algorithms are market-oriented, since the scheduler is not going to access

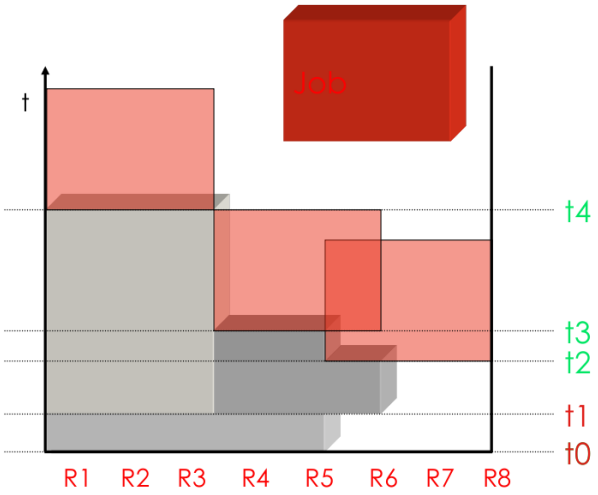

Figure 6. Example of offers creation in the Economic scheduling

the resource information but it simply evaluates the offers proposed by the resource providers as in an auction.

The figure 6 shows an example of offers creation. In the top right part of the figure we have the job that has to be executed. The shaded boxes are the offers collected. Clearly the most expensive is the one starting at t2 while the less expensive is the one starting at t4. The scheduler puts the costs into the objective function and choose the cheapest solution.

Very often instead of having something more simpler, we have something more complex because the users want to run jobs cheaply and as fast as possible. This leads to, as is called in mathematical terms, multi-criteria optimization. A typical objective function expressed by the users is:

$$util = U_{max} - (a_1 \cdot latency + a_2 \cdot price)$$

Market-oriented algorithms are considered robust, flexible, simple to adapt even if they can have unforeseeable dynamics.