Processing Identical Workflows on SOA Grids: Comparison of Three Approaches

Sékou Diakité, Jean-Marc Nicod, Laurent Philippe
LIFC/INRIA GRAAL, University of Franche-Comté
16 route de Gray, 25000 Besançon, France
[diakite,nicod,philippe]@lifc.univ-fcomte.fr

***Abstract*—In this paper we consider the scheduling of a batch of workflows on a service oriented grid. A job is represented by a directed acyclic graph without forks (intree) but with typed tasks. The processors are distributed and each processor have a set of services that carry out equivalent task types. The objective function is to minimize the makespan of the batch execution. Three algorithms are studied in this context: an on-line algorithm, a genetic algorithm and a steady-state algorithm. The contribution of this paper is on the experimental analysis of these algorithms and on their adaptation to the context. We show that their performances depend on the size and complexity of the batch and on the characteristics of the execution platform.**

***Keywords*—Batch scheduling, grid computing, heterogeneous platform, on-line scheduling, steady state scheduling, genetic algorithm.**

I. Introduction

Processing workflows of images is a very processor consuming activity therefore, when the size of the image set grows up it is mandatory to use a distributed execution platforms such as Grids. In this paper, the targeted application is a workflow that generates datas such as medical images. Each image is computed from a set of existing data on which we applied different operations: extracting data from an image, applying filters to enhance the image quality, calculating correlation between two images, merge of images to highlight some property and so on. After acquiring a set of images or data, we apply on them the same workflow, i.e. the same set of operations, to get the final result. However, the execution of such workflows on grids is not simple as the platform is heterogeneous from the performances viewpoint as well as from the deployed services viewpoint. Misplacement of execution may lead to very poor execution performances while certain medical applications need almost interactive results (not more than a few minutes). Thus, the schedule of the set of workflows must be carefully defined.

Different approaches can be used in this context to minimize the response time of such executions. Traditional scheduling techniques includes on-line approaches and off-line approaches. Our aim is to evaluate the efficiency of these approaches depending on the workflow and grid platform characteristics, then to use these results as input data to implement efficient scheduling policies in grid middlewares as DIET. As grid experiments are very costly to deploy and not repeatable due to the dynamic behavior of other applications, we started our evaluation by simulating the grid and the algorithms. The simulations have generated a first set of results presented here.

In this paper we compare the performances of three approaches. A simple on-line algorithm which schedules tasks depending on the computers and tasks availability and two off-line algorithms which are implemented to maximize the flow and makespan criterion. On one hand, we show that both the workflow's and platform's characteristics have an impact on the performances, this impact may be up to 50%, and, on the other hand, that the criteria to optimize must rather be makespan oriented when the number of workflows to perform is less than 100 and flow oriented when this number is above 250.

In the following section of this paper we define the application model, the platform's characteristics and the related works. In the third section we present the algorithms, their properties and different adaptations we did to well known algorithms to better fit our context. Then, in the forth section, we detail the implementation of these algorithms in a grid simulator. The fifth section is dedicated to the results and their analysis. A conclusion is given in the last section of the paper.

II. Context

Defining the context and the characteristics of the application execution is a very important step in the scheduling problem resolution. Indeed, scheduling researchers are actives since a long time and have addressed lots of scheduling issues. Therefore, the choice of scheduling algorithms must be done in regard to a particular platform model and a particular application model. In order to clearly define the platform's and application's characteristics, we first give a more formal model of the platform and of the workflows. Then we present the works related to the workflows scheduling on grids.

A. Platform model

Executing an application on the grid usually consist in asking a grid administrator, human or not, for nodes allocation. Then the system respond with a set of nodes that are availiable and the start time of the reservation. However, on Service Oriented Architecture Grids (SOA Grids) this model can vary as the request for nodes depends on the services deployed on the nodes. In some grid middlewares such as Diet or Ninf, the user asks for computing nodes that provides a set of services.

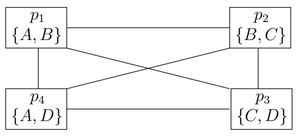


Fig. 1: Execution grid

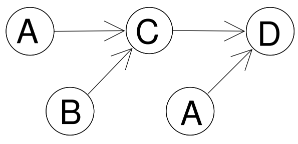


Fig. 2: Workflow

Different types of services can be installed on the same node. In this context, nodes are selected according to the services the workflow needs.

In our model, nodes and processors are equivalent, the execution platform is modeled as a set of processors, interconnected by network links. It is represented as an undirected graph: $PF = (P, L)$, where the vertices are the n nodes p_i ($p_i \in P : i \in [1..n], n = |P|$) and the edges are the network links l_i between these nodes.

Each node p_i implements a limited set of services or functions to carry out the different parts of the workflows. We define as F_i the set of functions that the processor p_i is able to perform.

Fig. 1 gives an example of a grid with the communication links. The communication graph is complete as the communications usually use the internet network. On this figure, the letters A, B, C and D represents the services deployed on the nodes of the grid; different services are deployed on the nodes.

TABLE I: Exec. platform performances

		p_1	p_2	p_3	p_4
Type	A	20	-	-	15
	B	10	10	-	-
	C	-	10	10	-
	D	-	-	10	10

Table 1 gives an example of the performances that characterize the platform. The given values correspond to the number of time units necessary to carry out a given service on a given processor, such values can be obtained by extrapolation of unary execution tests. When no value is given, the processor is not able to process the given service. For instance, processor p_4 can process functions A and D but cannot process functions B and C.

B. Application model

The defined platform model executes a batch of workflows. Usually a workflow is defined as a global task composed of sub-tasks. Each of these sub-tasks has to be executed in a constrained order. The workflow is done once all of it's subtask are computed.

Our target application is a set of images and data onto which we apply the same operations, the workflow, composed of filters or analysis operations. The whole set of images to be executed is available at some starting time `t` and the workflows are identical.

A batch B of length m is defined as a set of m identical instances J_j of the job (workflow) J such as $B = \{J_j : j \in [1..m]\}$. The job J is composed of several tasks with dependency constraints and is represented by a directed acyclic graph (DAG): $J = (T, D)$ where the vertices T are the tasks t_k and the edges D are the dependency constraints between the tasks. Note that there will be several instances of task t_k belonging to different instances J_j of job J.

Let F be the set of all the functions used by the job J thus $\cup_i F_i = F$ where $i \in [1..N]$. The time T needed to execute a task T_k on different nodes p_i is not uniform: $T(T_k, p_i) \neq T(T_k, p_i')$.

Fig. 2 gives an example of a job we have to schedule M times. The job is composed of five tasks. The arrows indicate the dependency constraints. Note that this job needs several time the same service A.

C. Related works

Scheduling problems have been studied extensively in the literature. In general, DAG scheduling is an NP-Complete problem [1].

[1] is a survey for scheduling DAG in a static context considering homogeneous platforms. Therefore, this problem is also NP-Complete in a Heterogeneous Computing (HC) environment. So, requiring of good heuristics is mandatory in this domain. Classical solutions to optimize the makespan of a set of tasks use heuristics such as Earliest Finish Time [2] or Critical Path [3]. A more complete study evaluates eleven heuristics in [4] for example: Min-min, Max-min or Sufferage. These 3 heuristics are based on the Minimum Completion Time (MCT). Genetic based scheduling algorithms [4], [5] gives good results in HC but these approaches are very time consumming compared to classical solutions. In the context of the Grid, [6] presents a modification of the 3 previous heuristics to take into account input and output data transfert times in the computation of MCT. [7] studies dynamic mapping heuristics considering a class of independent tasks in HC environment. When the tasks are multiple DAGs, problem of fairness between DAGs are studied in [8]. These approaches compute a off-line scheduling considering the whole set of tasks. If the number of tasks scales up, the computation time becomes too long due to the complexity of the algorithm. The strategies applied to schedule workflows onto the grid are presented in [9] with the description of real life medical application. Another example of a scientific application workflow is given in [10] about Ocean-Atmosphere modeling.

This study deals with static workflows scheduling in HC environment considering a few hundred of identical DAGs and limited resources. Finding an optimal schedule in this case is not possible with direct method. Two solutions are suitable in this context: (1) either use a scalable heuristic to compute a suboptimal schedule or (2) use the results of an optimal solution to a problem close to ours. So, three approaches match this context: makespan optimization heuristic using genetic algorithm, steady state techniques, and on-line techniques.

The suboptimal schedule strategy we evaluate is a static genetic based scheduling algorithm adapted to our context: GATS [11]. The algorithm explained the the next section is able to schedule a large number of DAGs.

The second approach optimize the throughput of the workflow in the HC environment. This Steady-state techniques achieve an optimal makespa n for an infinite number of identical jobs [12]. The resulting schedule is composed of an sub-optimal initialization stage that allow to enter the optimal steady-state stage and a sub-optimal termination stage that performs the remaining tasks. This schedule will tend towards optimality when the size of the batch increases, as the weight of the initialization and termination stages decreases in the global schedule. When the number of batches is too small, the sub-optimal stages overhead leads however to an inefficient schedule. So, the question is to evaluate for which number of batches the steady-State scheduling becomes interesting.

The third approach is an on-line oriented techniques that schedule tasks on the fly during the jobs execution. They take the state of the system into account to assign new tasks to processors. These techniques are very easy to implement and give rather good results. They give however no guarantee on the quality of the schedule.

In our context, we can note that only the Steady-State techniques benefit from the knowledge that the executed jobs are identical.

III. Selected solutions

Three algorithms are selected and adapted to our context characteristics to allow us to schedule a workflow of identical jobs. They represent different approaches and all of them give good results in our context. The first chosen algorithm is list-based on-line scheduling algorithm. This algorithm is used to give us a reference makespan to ease the comparison between the three algorithms. The second one is a static scheduling algorithm for graphs of aperiodic tasks on a heterogeneous platform. It is a genetic based approach that improves a list-based scheduling heuristic. It allows us to evaluate the performances of an algorithm designed for a set of tasks to schedule a workflow. The last algorithm we evaluate in this paper is a steady-state oriented scheduling algorithm well adapted to workflow and periodic jobs scheduling. A linear program allows us to optimize the throughput of the workflow regarding the constraints given by the platform and the periodic DAG to schedule. This solution is optimal in the steady-state period of time when the both initialization and termination stages are negligible [12]. It allows us to evaluate an algorithm initially designed for an unlimited workflow when the number of jobs are limited to a few hundred.

A. *On-line Scheduling with knowledge*

The workflow that we have to schedule is a set of DAGs but not a unique connected graph of tasks. Therefore, it is not possible to implement the well known EFT scheduling algorithm because it is not possible to identify the tasks with the earliest deadline: several tasks are ready at the same time but do not have any precedence constraint between them. So we implement a list-like scheduling algorithm which principle is to schedule a ready task as soon as possible on the processor that will perform it the first. No heuristic is used to select the task to schedule unlike a classical list based scheduling algorithms [2], [13]. This mechanism shortens the traversal time of a job. The best processor is selected using the Earliest Finish Time (EFT) heuristic [2].

B. *GATS*

Genetic Algorithm for Task Scheduling (GATS) [11] is a genetic based algorithm that improves about 7% to 10% a classical scheduling algorithm for aperiodic tasks on a heterogeneous platform. The GATS approach enhances the schedule computed first with a list based scheduling heuristic. The initial population is created at the first step: the first individual represents the result of a list-based schedule which favors the tasks on the critical path of J. Then one individual per processor represents a random schedule where all the tasks are affected to the given processor and the remain of the population represents random schedules.

GATS individuals represent tasks-processors allocations. Thus, the scheduling is computed by a decoding step of each individual. This reconstruction phase has to take into account the dependencies between tasks while respecting the allocation. The fitness f of each individual is deduced from the decoded schedule thanks to the formula: $f = 1/Makespan(schedule)$. The classical genetic operations on individuals such as mutations or cross-overs are easy to compute because of the simple chromosome representation without time informations and precedence constraints. Indeed, one task can be moved from one slot to another. Once the initial population is created, GATS performs a limited number of loops that compute the fitness of the individuals, a rank-based selection, mutations and cross-overs. The termination criterion used is this fixed number of loops. In the worst case, the computed schedule is at least equal to the initial list-based solution.

C. *Steady-state scheduling*

In our context, the optimization of the scheduling is to maximize the throughput of of the workflow we have to compute. But, we also have to respect the constraints due to the platform characteristics. These constraints ensure that jobs are fully computed. So, this optimization problem can be written as a linear program. When the system enters the steady-state stage, the solution is easy to compute because the linear program variables are rational thanks to the periodic scheduling approach. Indeed, the solution of this linear program gives the ratio of time spent by each processor for each task of the jobs and the proportion of time spent by each network link to send task results for each inter-tasks dependencies. This solution is translated into a weighted sum of allocation, where an allocation represents the traversal of a job in the platform (task/processor associations). The steady-state scheduling computes a period in which the allocations are interleaved according to their weight. The steady-state stage is the concatenation of the adequate number of these periods to compute the whole size of the workflow.

Before entering the steady state stage, the platform has to be initialized by computing every tasks needed to enter the steady-state. After the last period, a termination stage is also necessary to finish all the DAGs initiated by previous periods. In the original algorithm, the initialization is not optimized. The master computes itself every tasks without parallelism facility. As the initialization, the termination stage is performed by the master in a sequential way. Its role is to terminate every task staying in the platform after the last steady-state period.

IV. Simulation

The targeted applications are very time consuming. For this reason, we make the assumption that the bandwidth of PF allows us to neglect the communication time for all the computation steps of each instance J_i of the job J.

Building a mathematical model of the algorithms is not realistic due to the complexity of the problem. Their implementation on a grid cannot give reproducible results. So we use a grid simulator to evaluate the three algorithms.

The simulator is implemented above SimGrid and its MSG API [14], [15]. The algorithms are implemented with the master/slave paradigm. A master node runs the scheduling algorithms and dispatch the tasks to process to slaves node according to the scheduler decisions.

Slaves node runs two threads, the first one is in charge of receiving requests from others nodes. When the requests is a tasks computation, the task to perform is put on a FIFO queue. Other requests (for example query about queue length, time when the processor will be idle) are answered in place.

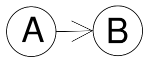
The second threads iteratively removes a task from the queue and execute it. The computing power of an host is not shared, tasks are executed with full computational resources one after another without preemption.

Two of the three algorithms are off-line scheduler, for them the simulator is just a tool to validate the computed schedule. Indeed an off-line scheduler has a full knowledge of the platform before the actual execution and it's computed schedule is just played back on the execution platform.

The third algorithm is an on-line scheduler, thus it is closely tied to the simulator. For each of it's scheduling decisions this scheduler has two query each of the processors for tasks queue length and the time when there will be able to finish a given task.

Two evaluate the simulated algorithms performances we introduce the notion of efficiency as the ratio between the optimal makespan over the simulated makespan. As the scheduling problem is NP-Hard and the size of the problem is important, we are not able to evaluate the optimal makespan. However the number of jobs to process (N) divided by the steady-state throughput (ρ) is a good optimal lower bound.

$$efficiency = makespan_o / makespan_r$$



		p_1	p_2
Type	A	10	10
	B	10	-

Fig. 3: Simple problem

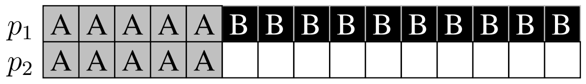


Fig. 4a: Gant diagram with infinite buffers

$$efficiency = N/(\rho \times makespan_r)$$

- $makespan_o$: lower bound
  - $makespan_o$: N/ρ,
  - reference time
- $makespan_r$: *makespan* resulting from experience,

V. Algorithms adaption

1) On-line: The on-line algorithm is a simple list scheduler. The task selection is performed by selecting the first task that is ready to be scheduled. The processor selection is based on the earliest finish time heuristic.

When scheduling multiple instances of the same job this approach is problematic. In the example of Fig. **??** the on-line algorithm produces the gant in Fig. 4a for 10 instances. All of the tasks A are scheduled first because at time 0 there are ready. Then at time 10 the first task B is ready but all the slot are reserved for A.

This schedule has a makespan of 150 ($5 \times A + 10 \times B$ on p_1) and during the last 100 time units, p_2 is idle. Our solution is to limits the size of the processors input buffer, for example with a limit of 1 the gant diagram is shown in Fig. 4b. Because of this buffer limit some tasks B are available concurrently with tasks A, the makespan is lower (110 time units) and processors p_2 is idle only for 20 time units.

2) GATS: The original GATS algorithm is not designed to schedule multiple instances of the same job. One simple method to make GATS able to solve this problem is to schedule all the job instances as a single job. This method increases however the time and the physical memory necessary for GATS to compute the schedule. So we use a hybrid method that schedules the tasks in successive intervals of x jobs and appends the set schedules.

This method introduces a new problem: the concatenation of two successive intervals leaves some processors idle so, we overlap intervals as shown on Fig. 5. The interval $Interval_i$

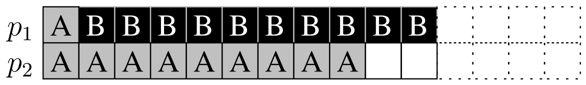


Fig. 4b: Gant diagram with buffers of size 1

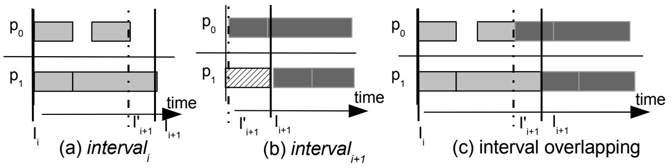


Fig. 5: Overlapping of $Interval_i$ and $Interval_{i+1}$.

ends when the processor P_1 finishes its task (fig.(a) I_{i+1}). The processor P_0 is idle between $I^{'}_{i+1}$ and I_{i+1} (fig. (b)). The interval overlapping re-uses this idle time in the next interval. The interval $Interval_{i+1}$ starts at $I^{'}_{i+1}$ instead of I_{i+1} with the information that P_1 is not available until I_{i+1}. Figure (c) shows the overlapping of $Interval_i$ and $Interval_{i+1}$.

3) Steady state: As explained in III-C, the objective of the steady-state algorithm is to maximize the throughput. A side effect is that the makespan-minimization for finite number of instances is not an objective. The original algorithm execute initialization and termination sequentially, in this study, we parallelize these non-optimal stages with the on-line algorithm.

VI. Algorithms comparison

In this part we compare the performances and the behavior of the chosen algorithms according to different parameters: platform characteristics, number of workflows to compute, etc. We study this behavior for different complexity and size of workflows.

The workflows are given by Fig. 6, Fig. 8 and Fig. 10. The letter inside the vertices of the workflow graph are the task type (i.e. the service) to be executed at this step of the workflow. The platforms are presented in Table 2. For instance, in this table, for the PF_3 platform, the processor p_4 needs 10 time units to perform a task A ($p_4(A) = 10$).

A. Small jobs

The first example is a very simple workflow composed of two tasks A and B where B use the result of A, see Fig. 6. We execute several instances of this job. The idea here is to show, using this very simple job, one of the drawbacks we have identified.

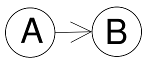


Fig. 6: Small job J_1

We execute this workflow on different homogeneous platforms, with equivalent execution time for the same function, or on heterogeneous platforms. The platforms described in Table 2 are the platforms used for Fig. 6 but we did more simulations to identify the possible issues. As a first result, we can note that even on homogeneous platforms the efficiencies of the algorithms may depend on the characteristics and very different results may be obtained as shown on Fig. 7a, Fig. 7b and Fig. 7c.

TABLE II: Simulation platform PF_1 to PF_3

Type	PF_1				PF_2			PF_3		
	p_1	p_2	p_3	p_4	p_1	p_2	p_3	p_1	p_2	p_3
A	10	-	100	1000	10	50	40	100	100	100
B	-	10	10	-	100	-	-	20	-	-

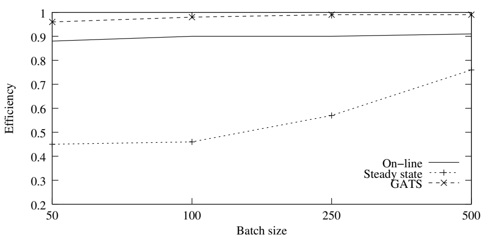


Fig. 7a: Simulation with platform PF_1 and job J_1

On these figures we can observe three different behaviors for the algorithms. First, on Fig. 7a GATS generates nearly optimal results which tends toward an efficiency of 1 around 250 workflows.

On the opposite, the steady-state gives poor efficiencies, two times less than the GATS when the number of jobs is less than 200. Then the efficiency slowly increases. The reason of these poor efficiencies is that this approach try to use as much as possible the whole platform capabilities to optimize the flow: so it uses p_4 for executing A tasks while it cost almost 100 times more than on other machines. To use p_4, the algorithm generate a very long period of 110 workflows, 220 tasks. Due to its conception, the algorithm first generates all the tasks needed to enter the steady-state period: 110 A tasks in the case of platform PF_1 and workflow J_1. This means that before entering the steady-state period the algorithm must perform 110 A tasks and, when the number of jobs is less than 110, the schedule does not start a period. Then, we have to execute 110 workflows more to realize a period. This means than under 220 workflows no complete period is realized. For sets of workflows bigger than 220, the steady-state will progressively makes up the difference with the other algorithms and tends toward optimal. So, a first drawback of this algorithm is that, by using all the resources available on the platform, it may generate poor schedules.

Another important remark on the steady-state schedule for the PF_1/J_1 case is that it starts by an initialization phase that generates all the needed tasks to enter the period. For these tasks, the aim is not to find the best schedule of the workflows implied in the initialization phase but rather to reach the period. We can illustrate that with the case of platform PF_1 and job J_1. Each period computed by the steady-state algorithm will generate 110 tasks. So to enter this period, we need to generate 110 A tasks that have precedence constraints with the 110 B tasks that will be executed in the period. Obviously, looking at the PF_1 platform, it does not lead to a good schedule to first executes all these A tasks. This is the reason why the schedule behaves so badly in this case.

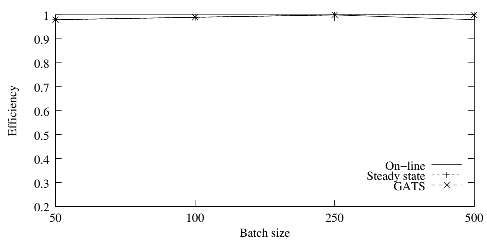


Fig. 7b: Simulation with platform PF_2 and job J_1

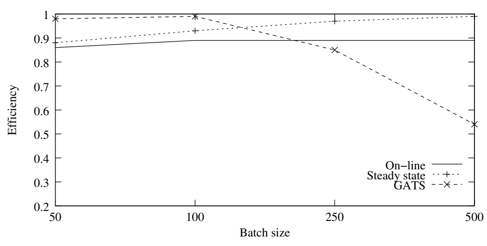


Fig. 7c: Simulation with platform PF_3 and job J_1

For the on-line scheduling, the efficiencies are stable around 0.9 which is rather good, but as good as for the GATS. The explanation is the short view of this algorithm: at each step it just schedules the ready tasks without taking the rest of the workflows that has to be done. So it does not benefit form a global view on the schedule. We can consider that this result is rather good compared to the simplicity of the algorithm.

On Fig. 7b, for platform PF_2 and the same job J_1, all the algorithms gets near optimal results. We can note that, in this case, the solution was obvious: the scheduling algorithms do not have an other choice than executing B tasks on processor p_1 and these tasks are critical as it it possible to execute A tasks more quickly either on processor p_2 or processor p_3. In this case, the short view of the on-line algorithm, is not a drawback as there is nothing else to do than executing as fast as possible these B tasks.

On Fig. 7c, for PF_3 platform and the same J_1 job, we can note that the GATS algorithm decreases its efficiencies when the number of workflows to execute increases. The on-line algorithm is stable almost the same as on Fig. 7a while steady-state algorithm quickly tends toward an optimal efficiency. The reason of these behavior is the choice that has to be done on processor p_1. To get good efficiencies, the three processors must be used to execute A tasks and sometimes processor p_1 must be used to execute B tasks. The steady-state algorithm find the optimal flow, with a period of 200 where p_1 start by executing 5 B tasks, for 100 and then execute one A while the other processors execute 2 A tasks each. In this case, GATS cannot find an optimal schedule when the size of the worksflows set is greater than 200. This results is closely linked to the size of the GATS period which is set to 400 tasks in this case, so 200 workflows. For the on-line algorithm, the results are explained as for Fig. 7a: the short view cannot anticipate to optimize.

In general, for small jobs, the scheduling decision is often simple and all the algorithms get good efficiencies, very close to the optimal. When the number of workflows is less that hundred, the GATS often gets the best efficiencies. However, the results are not reliable in the sense that on some particular values it may perform very badly. The standard list algorithm performs rather well but does not always reach the optimality: it may stay around 0.9 even when the number of workflows increase. The steady-state scheduling has to make up its initialization and termination stages.

From the three figures, Fig. 7a, Fig. 7b and Fig. 7c, we can also conclude that the architecture/characteristics of the platform impact on the algorithm behavior as, for the same simple job, the efficiencies may vary according to the following observations:

- the heterogeneity of the platform will extend the period and leads to longer initialization/termination phases and thus to an higher cost. So this algorithm will have no interest until a high number of workflows.
- the complexity of the platform may generate poor results for the GATS algorithm. This is probably linked to its division into intervals.
- the short view of the on-line algorithm may lower its efficiencies in some cases, in particular when the critical task must be executed on one processor and that other, earlier tasks, may be also affected to this processor.

In the following we study the behavior of the algorithms with much complex workflows.

B. Medium jobs

In this part we study the execution of jobs J_2 and J_3 (see Fig. 8 on the same platform PF_4 (see Table 3). The aim here is, on the one hand, to see if the behaviors differ when the complexity and size of the workflow is higher and, on the other hand, to analyze these behaviors regarding to the heterogeneity of the platform.

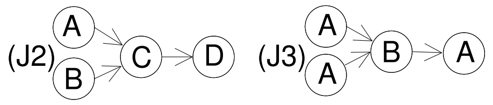


Fig. 8: Medium jobs

On the platforms given in Table 3 and compared to the platforms of Table 2 the heterogeneity is lower for both the execution times and the application available one the platforms.

TABLE III: Simulation platforms PF_4

Type	PF_4			
	p_1	p_2	p_3	p_4
A	20	-	-	20
B	10	10	-	-
C	-	10	10	-
D	-	-	-	10

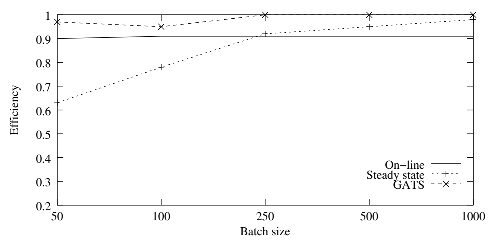


Fig. 9a: Simulation with platform PF_4 and job J_2

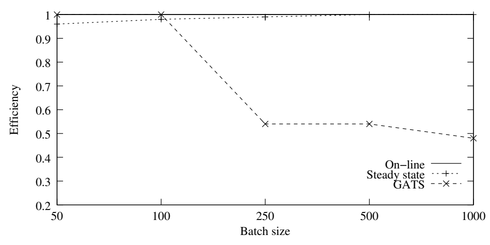


Fig. 9b: Simulation with platform PF_4 and job J_3

The results shown on Fig. 9a can be compared to Fig. 7a, where the steady-state slowly improve its efficiencies, and the results shown on Fig. 9b can be compared to Fig. 7c.

C. Big/complex jobs

This last part presents results with bigger jobs. To evaluate the impact of the dependency on the efficiencies, we use the same set of tasks but with different dependencies. Note that, as explained before, we have no communication cost in the experiments so that the deviations between the algorithms efficiencies is just linked to the dependencies characteristics.

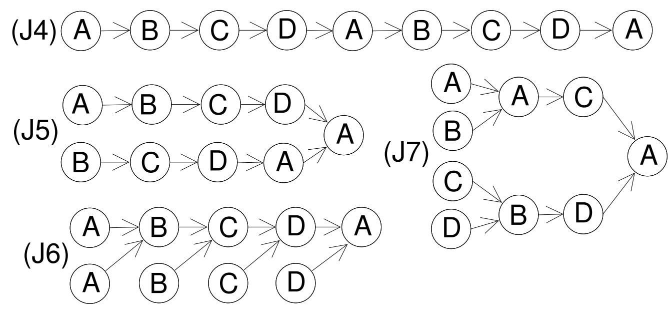


Fig. 10: Complex jobs

We execute these jobs on two different platforms PF_5 and PF_6 given in Table 4. Platform PF_5 is very simple and have homogeneous execution time. We added two more processors on platform PF_6. These processors have heterogeneous execution times for the different services.

Globally, the results in these cases are good as must of them are no more worst than 10% from the optimal. So the response of the algorithms to the complexity of the workflow is good. There no need to presents the results of the algorithms on platform PF_5 as all the algorithms find nearly optimal solutions for these workflows, whatever the number of workflows is. The difference between the algorithms results are limited to the initialization phase and are lower than 2% from the optimal.

TABLE IV: Simulation platforms PF_5 and PF_6

Type	PF_5				PF_6					
	p_1	p_2	p_3	p_4	p_1	p_2	p_3	p_4	p_5	p_6
A	10	-	-	-	10	-	-	-	100	1000
B	-	10	-	-	-	10	-	-	10	-
C	-	-	10	-	-	-	10	-	10	10
D	-	-	-	10	-	-	-	10	-	-

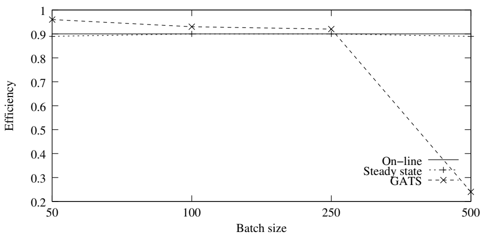


Fig. 11a: Simulation with platform PF_6 and job J_4

For platform PF_7, just adding two more processors gives very different results as shown on Fig. 11a and Fig. 11b. On Fig. 11b, all the algorithms performs well while on Fig. 11a, on-line and steady-state are 10% away from optimal, even for 500 jobs, and GATS decreases with the number of jobs.

The on-line algorithm and the steady-state algorithm performs equivalently due to the size of the steady-state period. The size of the period 100 and it terminates 37 jobs during this period. This means that the initialization phase have to execute a little bit less than 37×8 partial jobs before reaching the steady-state period. As a consequence, most of the 500 jobs are executed outside the period and the results are no so good.

For "chain like" jobs as J_4 and J_6, GATS performs well until 250 jobs but its efficiency collapses for bigger sets of jobs, while for J_7 jobs the results are close to the optimal (see Fig. 11b). For J_5, the efficiencies start to be unstable from 250 jobs. This probably depends on the interval size used. The explanation is in the length of the longest path in the workflow graph: the longest the path is the worst GATS behaves. When we have long path in the workflow's graph then the time spent by one workflow instance in the system increases. So the cost,

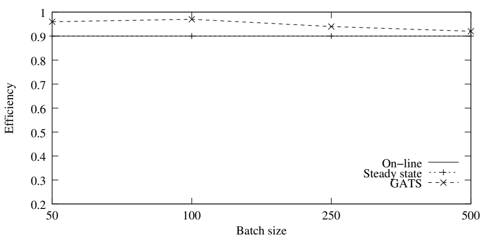


Fig. 11b: Simulation with platform PF_6 and job J_7

TABLE V: Schedule computation costs

Size	On-line scheduler	Steady-state	GATS
100	0.04s	0.05s	60.53s
200	0.10s	0.11s	85.46s
500	0.29s	0.23s	239.25s
750	0.79s	0.75s	439.25s
1000	1.09s	0.90s	617.43s

in term of idle time on processors, of using several intervals for GATS also increases. One solution to this problem could be to use bigger periods to limit these waste of time, however this increases the computing complexity of the schedule as shown is the next part.

D. Computation time of the schedules

In this part we compare the time costs to compute a schedule. As the workflows are executed on a grid, what we want is to execute them as soon as we get the resource or the nodes from the reservation service. So the time spent to compute the schedule is of importance as during this time we cannot execute the workflows.

The Table 5 shows the average time needed on an Intel Xeon running at 3.2 GHz to compute the schedules depending on the number of workflows to execute. Both the on-line algorithm and the steady-state algorithm are very fast to compute the schedule. GATS gets reasonable execution time for less than 200 jobs but clearly does not support on the fly execution above this value.

VII. Conclusion and future works

The paper presents a study on three different scheduling algorithms for a set of jobs on SOA grids. Our application context focus on workflows applied to data and a typical use case of this work is the processing of sets of data for medical analysis and image rendering. The originality of the work is, on the one hand, the schedule of identical jobs which allows to use algorithms designed for flow optimization and, on the other hand, the SOA grid context where not all applications or services are available on all servers. Each of the three compared algorithm is suited for a specific need: the on-line algorithm is simple, the steady-state algorithm guaranties an optimal flow and the GATS algorithm generates schedules with good makespan.

This study exhibit several results: the impact of the platform and of the workflow characteristics on the efficiencies and some drawbacks in the algorithms design. The on-line algorithm suffers from a short view and use processors that will be needed in the close future for critical tasks. The steady-state algorithm has too long sub-optimal stages when the platform is heterogeneous or when the workflows are complex. The GATS algorithm becomes costly and instable or generates poor schedules in some cases when the number of workflows increases.

Our future works will concentrate on improving these algorithms to fill these deficiencies. For the on-line algorithm, we will try to extend its view by defining a scheduling window where critical tasks could be identified. For the GATS algorithm, we clearly need to reduce its cost. A possible way is to better tune the genetic parameters and adapt the interval size depending on the platform. Finally, the steady state scheduling can be improved by optimizing the period size and the initialization stage.

References

[1] Y.-K. Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Task Graphs to Multiprocessors," *ACM Computing Surveys*, vol. 31, no. 4, pp. 406–471, Jan 1999.

[2] H. Topcuouglu, S. Hariri, and M. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," in *IEEE Trans. on Parallel and Distributed Systems*, 2002, pp. 260 – 274.

[3] Y. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multi-processors," in *IEEE Trans. on Parallel and Distributed Systems*, 1996, pp. 506 – 521.

[4] T. D. Braun, H. J. Siegel, N. Beck, L. L. Boloni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, and B. Yao, "A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems," *Journal of Parallel and Distributed Computing*, no. 61, pp. 810–837, 2001.

[5] L. Wang, H. J. Siegel, V. R. Roychowdhury, and A. A. Maciejewski, "Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach," *Journal of Parallel and Distributed Computing*, vol. 47, no. 1, pp. 8–22, Nov 1997.

[6] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman, "Heuristics for Scheduling Parameter Sweep Applications in Grid Environments," in *Heterogeneous Computing Workshop*, 2000, pp. 349–363.

[7] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing Systems," *Journal of Parallel and Distributed Computing*, vol. 59, no. 2, pp. 107–131, 1999. [Online]. Available: citeseer.ist.psu.edu/article/maheswaran99dynamic.html

[8] H. Zhao and R. Sakellariou, "Scheduling multiple DAGs onto heterogeneous systems," in *HCW06*, Rhodes, Greece, Apr 2006, p. 14.

[9] A. Mandal, K. Kennedy, C. Koelbel, G. Marin, J. Mellor-Crummey, B. Liu, and L. Johnsson, " Scheduling strategies for mapping application workflows onto the grid," in *Proceedings. 14th IEEE International Symposium on High Performance Distributed Computing, 2005. HPDC-14.*, NC, Triangle Park, USA, Jul 2005, pp. 125–134.

[10] Y. Caniou, E. Caron, G. Charrier, A. Chis, F. Desprez, and M. Eric, "Ocean-Atmosphere Modelization over the Grid," in *The 37th International Conference on Parallel Processing (ICPP 2008)*, 2008, to appear.

[11] M. Daoud and N. Kharma, "Gats 1.0: A novel ga-based scheduling algorithm for task scheduling on heterogeneous processor nets," in *Genetic And Evolutionary Computation Conference*, 2005.

[12] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert, *Assessing the impact and limits of steady-state scheduling for mixed task and data parallelism on heterogeneous platforms*. IEEE Computer Society Press, 2004.

[13] M. Wu, W. Shu, and H. Zhang, "Segmented min-min: A static mapping algorithm for meta-tasks on heterogeneous computing systems," in *9th Heterogeneous Computing Workshop*, 2000, pp. 375 – 385.

[14] H. Casanova, "Simgrid: A toolkit for the simulation of application scheduling," *ccgrid*, vol. 00, p. 430, 2001.

[15] H. Casanova, A. Legrand, and L. Marchal, "Scheduling distributed applications: the simgrid simulation framework," in *3rd IEEE Intl Symposium on Cluster Computing and the Grid*, 2003.