Parallel Processing Letters

Practical Steady-State Scheduling for Tree-Shaped Task Graphs

Sékou Diakité[1], Loris Marchal[2], Jean-Marc Nicod[1] and Laurent Philippe[1]

1:Laboratoire d'Informatique de Franche-Comté
Université de France Comté, France

2:Laboratoire de l'Informatique du Parallélisme
CNRS - INRIA - Université de Lyon, France

Abstract

In this paper, we focus on the problem of scheduling a collection of similar task graphs on a heterogeneous platform, when the task graph is an intree. We rely on steady-state scheduling techniques, and aim at optimizing the throughput of the system. Contrarily to previous studies, we concentrate on practical aspects of steady-state scheduling, when dealing with a collection (or *batch*) of limited size. We focus here on two optimizations. The first one consists in reducing the processing time of each task graph, thus making steady-state scheduling applicable to smaller batches. The second one consists in degrading a little the optimal-throughput solution to get a simpler solution, more efficient on small batches. We present our optimizations in details, and show that they both help to overcome the limitation of steady-state scheduling: our simulations show that we are able to reach a better efficiency on small batches, to reduce the size of the buffers, and to significantly decrease the processing time of a single task graph (latency).

Keywords: static scheduling, grid computing, steady-state scheduling

1. Introduction

Computing Grids gather large-scale distributed and heterogeneous resources, and make them available to large communities of users [8]. Such platforms enable large applications from various scientific fields to be deployed on large numbers of resources. These applications come from domains such as high-energy physics [4], bioinformatics [13], medical image processing [10], etc. Distributing an application on such a platform is a complex duty. As far as performance is concerned, we have to take into account the computing requirements of each task, the communication volume of each data transfer, as well as the platform heterogeneity: the processing resources are intrinsically heterogeneous, and run different systems and middlewares; the communication links are heterogeneous as well, due to their various bandwidths and congestion status.

Applications are usually described by a (directed) graph of tasks. The nodes of this graph represent the computing tasks, while the edges between nodes stand for the dependencies between these tasks. These dependencies are usually materialized by files: a task produces a file which is necessary for the processing of some other tasks. In this paper we consider *Grid workflows* made of a collection of input data sets that must all be processed

by the same application. We thus have several instances of the same task graph to schedule. Such a situation arises when the same computation must be performed on independent data [11] or independent parameter sets [15]. Moreover, the targeted applications that we plan to schedule do not include any replication phases in the process. Hence, DAGs considered in this paper have no fork nodes, and consists of chains or in-trees. This corresponds to application like medical [12] or media [14] image processing workflows.

The problem consists in finding a schedule for these task trees which minimizes the overall processing time, or makespan, on a dedicated heterogeneous computational environment. This problem is NP-hard as it is a generalization of the $P||C_{max}$ problem which is NP-hard [9]. To overcome this issue, the work presented in [1] proposed to use steady-state scheduling. In this relaxation of the problem, the instances to be performed are assumed to be so numerous that after some initialization phase, the flow of computation will become steady in the platform. By characterizing resource activities in this *steady state*, it is possible to derive a periodic schedule that maximizes the *throughput* of the system, that is the number of task graphs completed within one time unit. As for makespan minimization, this schedule is asymptotically optimal. This means that for a very large number of instances to process, the initialization and clean-up phases that wrap the steady-state phase become negligible, and the makespan of the steady-state schedule becomes close to the optimal one. However, when the number of instances is important but bounded, existing steady-state approaches do not give optimal performance – initialization and clean-up phases cannot be neglected when scheduling a finite number of instances – and lead to a huge number of ongoing instances. In the present paper, we propose an adaptation of the steady-state scheduling that makes it suitable to batches of task graphs of medium size, that is consisting of a few hundreds of task graphs.

The rest of the paper is organized as follows. In Section 2 we give a short reminder on the steady-state techniques and their drawbacks. Sections 3 and 4 present our optimizations for practical settings, and Section 5 reports the simulations which have been performed to establish the relevance of our optimizations.

2. Steady-state scheduling for task graphs

2.1. *Platform and application model*

In this section, we detail the model used in the following study. First, we denote by $G_P = (V_P, E_P)$ the undirected graph representing the platform, where $V_P = \{P_1, \ldots, P_p\}$ is the set of all processors. The edges of E_P represent the communication links between these processors. The time needed to send a unit-size message between processors P_i and P_j is denoted by $c_{i,j}$. We use a bidirectional one-port model: if processor P_i starts sending a message of size S to processor P_j at time t, then P_i cannot send any other message, and P_j cannot receive any other message, until time $t + S \times c_{i,j}$.

The application is represented by a directed acyclic graph (DAG) $G_A = (V_A, E_A)$, where $V_A = \{T_1, \ldots, T_n\}$ is the set of tasks, and E_A represents the dependencies between these tasks. A dependency $T_k \rightarrow T_l \in E_A$ means that there is a file $F_{k,l}$ produced by task T_k and consumed by task T_l. We use an unrelated computation model: computation of task

T_k needs a time $w_{i,k}$ to be entirely processed by processor P_i. The tasks of V_A are typed and every processor is only able to compute a subset of the task types.

We assume that we have a large number of similar task graphs to compute. Each instance is described by the same task graph G_A, but has a different input file from the others. This is the case when the same computation has to be performed on different input data sets.

2.2. *Principle*

The present study is based on a steady-state approach for scheduling collections of identical task graphs proposed in [1]. In this section, we briefly recall steady-state techniques and their use for task graph scheduling. The steady-state approach has been pioneered by Bertsimas and Gamarnik [2]. The main interest of this approach is to optimize the throughput of the system instead of the classical makespan. It is based on a linear programming approach that computes a schedule period to provide an optimal solution to the problem. Some steps of the schedule construction are however complex, especially for handling communications, and we refer the interested reader to this article for a more detailed description.

The steady state is characterized using activities variables: α_i^k represents the average number of tasks T_k processed by processor P_i within one time unit in steady state. We similarly define activities for data transfers: $\beta_{i,j}^{k,l}$ represents the average number of files $F_{k,l}$ sent by P_i to P_j within one time unit in steady state. Then, we write constraints which express the limited capacity of the processors and links, on these activity variables. We also write "conservation laws" to state that files $F_{k,l}$ have to be produced by tasks T_k and are necessary to the processing of tasks T_l. We obtain a set of constraints which depicts a valid steady-state schedule, described using α and β variables. We add the objective of maximizing the throughput, that is the overall number of task graphs processed per time unit, which can be computed as $\min_k \sum_i \alpha_i^k$, and we get a linear program. Solving this linear program over the rational numbers allows us to compute the optimal steady-state throughput, and the description of an optimal solution.

The optimal solution obtained with this linear program describes a schedule through activity variables. This description is not very handy, and we need to precisely state where and when is done each task for each task graph of the series. We construct a periodic schedule that achieves an optimal throughput. This is done in two steps. First, we construct a set of *allocations*: an allocation is a mapping of all tasks of the task graph on the processors. For example, Figure 1 shows two possible allocations of a simple task graph on a platform with three processors. Allocation A_1 maps task T_1 on processor P_1 and task T_2 on processor P_3, while allocation A_2 maps task T_1 on processor P_2 and task T_2 on processor P_3. In general, an optimal steady-state schedule consists of a weighted sum of allocations: each allocation is provided with a throughput, and several allocations are used simultaneously. For example, in Figure 1, both allocations have a throughput of one task graph per period, thus resulting in a total throughput of two task graphs processed per period.

Then, we have to state when each transfer and each computation is scheduled in a period of the periodic schedule. In the solution of the linear program, the average number of tasks

(or files) processed (or transfered) in a time unit may be rational. However, we cannot split the processing of a task, or the transfer of a file, into several pieces. Thus, we compute the lowest common multiple LCM of all denominators of these quantities. We then multiply all quantities by LCM, to get a period where every quantity of tasks or files is integer. So, although bounded, the length L of the period may be large. A period describes the activity of each processor (how many tasks of each type is performed) and of each link: communications are assembled into groups that can be scheduled simultaneously without violating the one-port model constraints. In the following, we consider these communication groups as one special task, assigned to a fictitious processor P_{p+1}; a dependency between a task T and a file F is naturally transformed into a dependency between T and the special task representing the group of communication which contains the file transfer F.

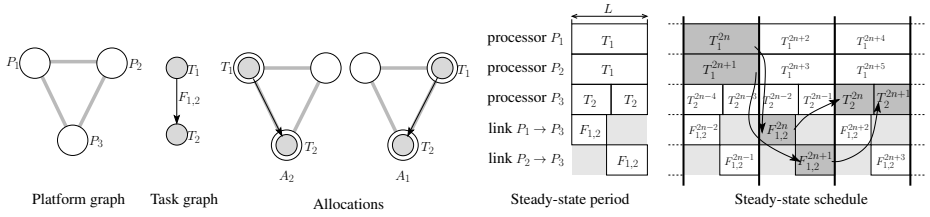


Figure 1. Example of steady-state allocations, period, and schedule.

The steady-state schedule is made of a pipelined succession of periods, as described in Figure 1. Dependencies between files are taken into account when reconstructing the schedule: a file $F_{k,l}$ produced by T_k during period 1 will be transferred to another processor during period 2 and then used by task T_l during period 3. Figure 1 describes a steady-state schedule obtained for a simple task graph: in a period both processors P_1 and P_2 process a task T_1, while P_3 processes two tasks T_2, achieving a throughput of 2 instances every L time units. In the periodic schedule, each task or file transfer is provided with its instance number in superscript, and dependencies are materialized with arrows for instances $2n$ and $2n+1$.

Once the periodic schedule is built, it can be used to process any number of tasks. A final schedule consists of three phases:

(1) an initialization phase, where the preliminary results needed to process the first period are pre-computed;
(2) the steady-state phase, composed of several periods;
(3) a clean-up phase, where all remaining tasks are processed so that all instances are completed.

Note that the problem of constructing a steady-state schedule that maximize the throughput has a polynomial complexity when restricting to DAGs with *bounded dependency depth* (which covers many important cases, such as trees), and that the problem for general DAGs is NP-complete [1]. Since the current study is limited to in-trees, it is possible to use this method to build a optimal steady-state schedule.

2.3. *Shortcomings*

We have seen that the length L of the period of the steady-state schedule may be quite large, and that a large number of periods may be needed to process a single task graph in steady state. This induces a number of drawbacks:

Long latency. For a given task graph, the time between the processing of the first task and the last task, also called latency, may be large since several periods are necessary to process the whole instance. This may be a drawbacks for interactive applications.

Large buffers. Since the processing time of each instance is large, a large number of instances must be started before the first one is completely processed. Thus, at every time step, a large number of ongoing jobs have to be stored in the system, and the platform must provide large buffers to handle all temporary data.

Long initialization and clean-up phases. Since the length of the period is large and contains many task graph instances, the number of tasks that must be processed before entering steady state is large: for each dependency of the initial period the sub-graph that creates the corresponding file it must be processed. Thus, the initialization phase will be long. Similarly, after the steady-state phase, many tasks remain to be processed to complete the schedule, leading to a long clean-up phase. As these phases are done using an heuristic scheduling algorithms, their execution time might be far from the optimal, leading to poor performance of the overall schedule.

In spite of these drawbacks, we have shown in [6] that steady-state scheduling is of practical interest as soon as the number of task graph instances is a few thousands. In the present study, we aim at overcoming the aforementioned shortcomings of steady-state scheduling, in order to get efficient schedules for batches of medium size, that is, batches containing a few hundreds of task graphs. The overall metric is the time needed to process all the DAGs (total *makespan*). However, by using steady-state scheduling, we also focus on *throughput* maximization.

In this paper, we focus on both reasons of these drawbacks. First, in Section 3, we propose a solution to reduce the number of periods necessary to process one instance, by simplifying the dependency scheme of a period. Then, in Section 4, we aim at shortening the period, at the cost of a small reduction in the system throughput.

3. Minimizing the number of inter-period dependencies

3.1. *Motivation*

We aim at scheduling identical DAGs (in-trees) on a heterogeneous platform with unrelated machines. Our typical workload consists of a few hundreds of DAGs. When the period of the steady-state is small compared to the length of the steady-state phase, initialization and clean-up phases are short, and steady-state scheduling is a very good option. When the period obtained is large compared to the steady-state phase, it is questionable to use steady-state as initialization and clean-up may render its advantage unprofitable. Thus, the *length*

of the period is a key parameter for our objective. In this section, since we want to keep an optimal throughput, we do not reduce this length. However, we prohibit any increase in the period length, which would go against our final objective.

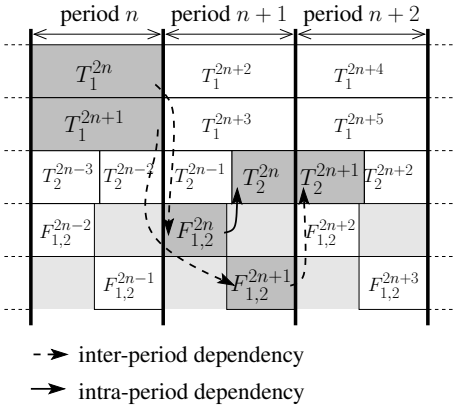


Figure 2. A periodic schedule with inter-period and intra-period dependencies

We have seen that a large number of periods may be needed to completely process one instance. More precisely, after building the steady-state period, each dependency in the task graph can be satisfied within a period, or between two consecutive periods, as illustrated in Figure 2. In the figure, we have taken the period of Figure 1, and we have modified its utilization of the schedule, so that one dependency can be satisfied within a period: in the new schedule, the results of file transfer $F_{1,2}$ can be used by task T_2 immediately, in the same period, instead of waiting for the next period. This is done by reorganizing the period: the "first" transfer $F_{1,2}$ of a period is now used to compute the "second" task T_2. We say that $F_{1,2} \rightarrow T_2$ is an *intra-period* dependency, contrarily to other dependencies that are *inter-period*. Of course, this single modification has little impact on the total makespan, but if we could transform all inter-period dependencies into intra-period dependencies (or a large number), our objective would be greatly improved.

The *number of inter-period dependencies*, that is the dependencies which originate in one period and terminate in the following one, is an important factor. The number of periods needed to completely process an instance (and thus the latency) strongly depends on the number of such dependencies. As for the makespan, the number of instances that have to be started in the initialization phase, and finished in the clean-up phases is exactly the number of inter-period dependencies. So when an *inter-period* dependency is transformed to an *inter-period* dependency, the length of the sub-optimal initialization and clean-up phases are reduced in favor of the optimal steady-state phase. Thus, reducing the number of *inter-period* dependencies is an important goal in order to overcome the drawbacks of the original steady-state implementation. Note that in the original version of the steady-state schedule, the number of these dependencies is huge: all dependencies are inter-period dependencies.

3.2. *Formalization of the problem*

We start from the description of the period obtained with the steady-state algorithm. A period consists of q instances of the task graph G_A. The u^{th} instance of task T_k is denoted T_k^u. We call $\sigma(P_i)$ the set of instances of tasks processed by processor P_i. For sake of simplicity, we denote by w_k^u the duration of T_k^u, that is $w_k^u = w_{i,k}$, with $T_k^u \in \sigma(P_i)$.

Dependencies between task instances naturally follow the edges of the task graph: for each edge $T_k \to T_l \in E_A$, for all $u = 1, \ldots, q$, we have a dependency $T_k^u \to T_l^u$.

The period is provided with a length L, which must not be smaller than the occupation time of any processor: $\sum_{T_k^u \in \sigma(P_i)} w_k^u \leq L$ for all P_i.

The solution to our problem consists in starting times $t\left(T_k^u\right)$ for each instance of task T_k^u. We must ensure that two tasks scheduled on the same processor do not overlap:

$$\begin{aligned} \forall P_i,\ \forall T_k^u, T_l^v \in \sigma(P_i),\ \text{with } t\left(T_k^u\right) \neq t\left(T_l^v\right), \\ t\left(T_k^u\right) \leq t\left(T_l^v\right) \Rightarrow t\left(T_k^u\right) + w_k^u \leq t\left(T_l^v\right) \end{aligned} \tag{1}$$

The number of inter-period dependencies for a given solution can be easily computed. A dependency $T_k^u \to T_l^u$ is an intra-period dependency if and only if T_k^u finishes before the beginning of T_l^u, that is if

$$t\left(T_k^u\right) + w_k^u \leq t\left(T_l^u\right) \text{ with } T_l^u \in \sigma(P_i). \tag{2}$$

Thus, inter-period dependencies are all dependencies that do not satisfy this criterion.

3.3. *Complexity of the problem*

In this section, we assess the complexity of the problem presented in the previous section, namely the ordering of the tasks on each processor, with the objective of minimizing the number of inter-period dependencies.

We first define the decision problem associated to the minimization of the number of inter-period dependencies.

Definition 3.1 (INTER-PERIOD-DEP) *Given a period described by its schedule σ and its length L, consisting of q instances of a task graph G_A (which is a tree), on p processors, with computation times given by w, and an integer bound B, is it possible to find starting times $t\left(T_k^u\right)$ for each task instance such that the resultant number of inter-period dependencies is not larger than B?*

It turns out that this problem is NP-complete. The proof of this result, based on a reduction from the 3-PARTITION problem, is available in the companion research report [7].

We now present two solutions for the problem presented in Section 3.2. The first solution uses a linear program approach that makes use of both integer and rational variables to compute an optimal solution, hence it is a Mixed Integer Program. Solving a MIP is NP-complete, however efficient solvers exist for this problem [5], which makes it possible to solve small instances. In the second solution, we first construct a period schedule without taking dependencies into account, and then use a greedy algorithm to find the maximum number of intra-period dependencies. Since the period schedule is constructed beforehand

and the algorithm is not allowed to move tasks when looking for dependencies, this approach is only heuristic.

3.4. *Optimal algorithm with MIP formulation*

In the following, we assume that we have only one instance of the task graph in the period, for sake of readability. Furthermore, we denote by w_j the processing time of T_j on the processor which executes it. Our approach can be extended to an arbitrary number of instances, at the cost of using more indices.

For any pair of tasks (T_j, T_k) executed on the same processor (that is such that $T_j, T_k \in \sigma(P_i)$ for some P_i), we define a binary variable $y_{j,k}$. We will ensure that $y_{j,k} = 1$ if and only if T_j is processed before T_k.

We also add one binary variable $e_{j,k}$ for each dependency $T_j \to T_k$. This binary variable expresses if the dependency is an intra-period dependency ($e_{j,k} = 1$) or an inter-period dependency ($e_{j,k} = 0$).

Finally, we use the starting time t_j of each task T_j as a variable. We now write constraints so that these variables describe a valid period.

- We ensure that the y variables correctly define the ordering of the t_j's:

$$\forall P_i, \forall T_j, T_k \in \sigma(P_i), \quad t_j - t_k \geq -y_{j,k} \times L \tag{3}$$

$$y_{j,k} + y_{k,j} = 1 \tag{4}$$

- We check that all tasks are processed within the period:

$$\forall T_j, \quad t_j + w_j \leq L \tag{5}$$

- We also check that if $e_{j,k} = 1$, the corresponding dependency is intra-period:

$$\forall T_j \to T_k, \quad t_k - (t_j + w_j) \geq (1 - e_{j,k}) \times L \tag{6}$$

- Finally, we make sure that no task is processed during the processing of task T_j, that is during $[t_j, t_j + w_j]$:

$$\forall P_i, \forall T_j, T_k \in \sigma(P_i), T_j \neq T_k, \quad t_k - (t_j + w_j) \geq (y_{j,k} - 1) \times L \tag{7}$$

Together with the objective of minimizing the number of inter-period dependencies (i.e., maximizing the number of intra-period dependencies), we get the following MIP:

$$\begin{cases} \text{Maximize } D = \sum e_{j,k} \\ \text{under the constraints } (3), (4), (6), (7) \text{ and } (5) \end{cases} \tag{8}$$

We can prove that the previous linear program computes a valid schedule with a minimal number of inter-period dependencies (see the companion research report for details [7]).

3.5. *Greedy approach*

The major difference between the previous MIP approach and the greedy approach that we describe here is the management of the instance indices. In the MIP approach, all instances are distinguished, and the previous linear program is in fact written with T_j^u variables, u being the index of the instance. In the above study, we have discarded this u index simply to get lighter notations, but the MIP clearly separates tasks of different instances.

In the greedy approach, we contrarily merge all tasks of the same type coming from different instances: all tasks T_j are mixed whatever the real instance T_j^u. In order to get a real schedule, with correct instances, we will reconstruct the complete task graph for each instance later, at the end of this phase.

After merging all instances of the same task, we get several occurrences of the same task on each processor. We first decide the processing order of every occurrence on each processing element for one period. This is done with the help of a simple one-dimensional load-balancing algorithm. As a result, all tasks will be optimally distributed in the period. For example, if a processor has to execute three occurrences of task A and three occurrences of task B, we will produce a schedule ABABAB, or BABABA, instead of AAABBB.

Once these local schedules have been constructed, we decide not to move tasks anymore, contrary to what the MIP approach does. We then intent to maximize the number of intra-period dependencies. To this goal, consider any dependency $T_k \rightarrow T_l$. Occurrences of T_k might be allocated (and now scheduled) on several processors, and the same holds for occurrences of T_l, but there are as many occurrences of T_k as T_l. All results of tasks T_k are needed for the processing of tasks T_l. A given occurrence of T_l can only use the results of an occurrence of T_k that was processed earlier. We thus use a greedy algorithm to connect a maximal number of tasks T_l to a predecessor T_k using an intra-period dependency: for the first occurrence of task T_k, we denote by t its completion time, we select the first occurrence of T_l that starts after time t, if it exists, and we allocate the intra-period dependency between these two occurrences. We suppress these occurrences from our list and continue until there is no more possible intra-period dependency. All remaining dependencies are allocated as inter-period dependencies.

4. Using non-conservative steady-state solutions to improve efficiency

4.1. *Motivation*

As we have seen that many of the steady-state scheduling drawbacks come from an excessive period length, we aim in this section at reducing the length of the period. However, we face a major issue: the length of the period, as well as its composition, is dictated by the solution of a linear program, as explained in [1]. Thus, we have very little hope to find a schedule with shorter period reaching the optimal throughput. Here, we choose to modify the solution, and to potentially decrease the system throughput, in order to gain flexibility on the period length. Our claim is that we can significantly shorten the period at the cost of a slight reduction of the throughput, which will result in a speed-up for medium-size batches, as well as in more practical schedules, with shorter latencies and smaller buffers.

4.2. *Principle and algorithm*

As described above in Section 2.2, a steady-state schedule consists of a superposition of allocations $A_1, \ldots, A_m$. Allocation A_k has throughput ρ_k, such that the whole schedule reaches throughput $\rho = \sum_k \rho_k$. Each ρ_k is a rational number, which is written α_k/β_k, where α_k and β_k are relatively prime integers. In order to process (and transfer) an integer number of tasks (and files) during a period, the least common multiple of all denominators ($T = \text{lcm}_k \beta_k$) is chosen as the length of the period. In one period, allocation A_k processed $T \times \alpha_k/\beta_k$ tasks, which is integer. However, some β_k may be large, leading to a large period length T. It may even happen that the corresponding throughput α_k/β_k is small, that is, allocation A_k contributes to a small amount to the total throughput, but is responsible for the long period. Such allocations are needed to reach the optimal throughput, but they are a pitfall for medium-size batches. Thus, we choose to suppress them and we claim that the loss in the throughput will be compensated by the shorter period, which reduces the initialization and clean-up phases, and make the steady-state scheduling more efficient.

Indeed, given an allocation A_k with a large β_k, we do not necessary want to suppress it (that is, to subtract its throughput ρ_k from the total throughput). We simply suppress β_k from the computation of the new period length T', and we scale down its throughput to $\lfloor (\alpha_k \times Tt)/\beta_k \rfloor$.

Algorithm 1: Shorten the period of a steady-state schedule.

Data: Total number of instances N_{total}, and a solution described by m allocations, allocation i has throughput α_i/β_i.
Parameters: K (maximum relative weight of initialization phase) and L (maximum throughput degradation).
Sort allocation by non-increasing β_k, so that $\beta_1 \geq \beta_2 \geq \cdots \geq \beta_m$
$N_{init} \leftarrow$ estimateInitTermJobCount$(\rho_1, \ldots, \rho_m)$
$i \leftarrow 1$
$\rho^{orig} = \sum_{k=1}^{m} \alpha_k/\beta_k$
while $i < m-1$ *and* $(N_{init}/N_{total} > K)$ *and* $(\rho > L \times \rho^{orig})$ **do**
  $T \leftarrow \text{lcm}\{\beta_i, \ldots, \beta_m\}$
  foreach *allocation* A_k *in* $\{A_1, \ldots A_m\}$ **do**
    $\rho_k^{\text{rollback}} \leftarrow \rho_k$
    $\rho_k \leftarrow \lfloor (\alpha_k \times T)/\beta_k \rfloor$
  $\rho \leftarrow \sum_{k=1}^{m} \rho_k$
  $N_{init} \leftarrow$ estimateInitTermJobCount$(\rho_1, \ldots, \rho_m)$
  $i \leftarrow i + 1$
if $(N_{init}/N_{total} \leq K)$ *or* $(\rho \leq L \times \rho^{orig})$ **then**
  foreach *allocation* A_k *in* $\{A_1, \ldots A_m\}$ **do** $\rho_k \leftarrow \rho_k^{\text{rollback}}$
return $(\rho_1, \ldots, \rho_m)$

Our approach is summarized in Algorithm 1. This algorithm relies on the estimateInit-

TermJobCount function to estimate the number of partial task graphs which have to be started during the initialization (and must be finished during the termination phase) for a given allocation. This allows us to estimate the relative duration of the initialization and termination phases compared to the steady-state phase. Our objective is that the steady-state phase becomes dominant. More specifically, we introduce a parameter K to upper bound the ratio of the number of task graphs started in the initialization to the total number of task graphs. We also bound the degradation of the throughput using another parameter L. In the simulations, we tested a large range of values for these parameters, but we report the results only for the best choice, namely $K = 0.25$ (at least three fourth of the task graphs should be processed in steady state), and $L = 0.85$ (we tolerate a maximum decrease of the throughput of 15%).

5. Experimental results

In this section, we present experimental results that show how minimizing the inter-period dependencies and/or how using the non-conservative period reduction improves the original steady-state algorithm. We compare six algorithms that schedule batches of task graphs on a heterogeneous platform:

- The original steady-state implementation proposed in [1];
- The steady-state approach improved with the reduction of inter-period dependencies through Mixed Integer Programming (see Section 3.4), denoted by steady-state+MIP;
- The steady-state approach improved with the reduction of inter-period dependencies using the greedy heuristic (see Section 3.5), denoted by steady-state+heuristic;
- The steady-state approach improved using the non-conservative period reduction (see Section 4), denoted by steady-state+suboptimal;
- The steady-state improved with both the reduction of inter-period dependencies using the greedy heuristic and the non-conservative period reduction, denoted by steady-state+heuristic+suboptimal;
- A classical list scheduling algorithm based on HEFT (Heterogeneous Earliest Finish Time, see [16]): as soon as a task or a communication is freed of its dependencies, the algorithm schedules it on the resource that guarantees the earliest finish time. Its evaluation depends on the load of the platform and takes both the computation time and the communication time into account. Note that in all steady-state strategies, the initialization and clean-up phases are implemented using this list-scheduling technique.

5.1. *Simulation settings*

The experiments were performed with a simulator implemented above SimGrid, using its MSG API [3]. The simulations consist of 200 platform/application scenarios for batches counting from 1 to 10 000 task graphs. Platforms are randomly generated, and comprise between 4 to 10 nodes. Tasks are gathered into 10 types, and each processor is able to

process a subset of the 10 types. The computation times range between 1 and 11 time units. Network links form a homogeneous graph connecting all computing nodes. Applications are randomly-generated in-trees of 5 to 12 tasks selected among the different types. The size of the files associated to dependencies are slightly heterogeneous (either one or two units of size).

For some scenarios, large periods and large numbers of dependencies may arise, so that solving the Mixed Integer Program is not possible, even though we use an efficient MIP solver (CPLEX [5]). In the following, we thus distinguish two cases: SIMPLE scenarios are the ones when we are able to solve the MIP (142 scenarios), and GENERAL scenarios gathers all cases (200 scenarios).

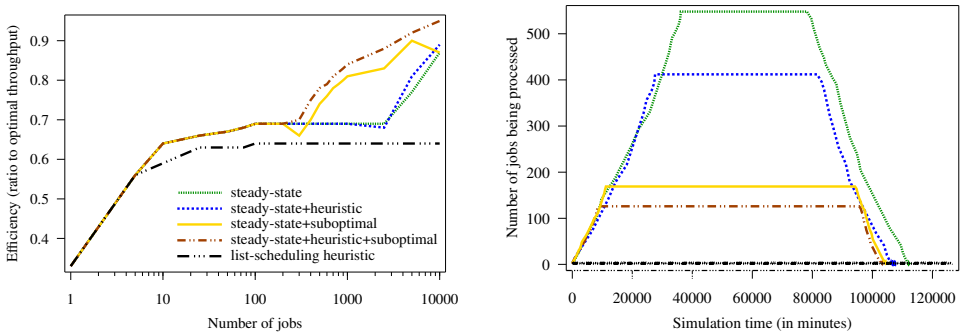
5.2. *Impact of the reduction of the inter-period dependencies*

In order to estimate the efficiency of the first optimization, we compute the number of inter-period dependencies that the different strategies (MIP or heuristic) are able to transform into intra-period dependencies. When we are able to solve the MIP, it suppresses 32% of the inter-period dependencies, whereas the heuristic is able to suppress 25% of them (26% in SIMPLE cases). This shows that both the MIP and the heuristic strategies achieve a good performance for this metric. As we have outlined before, this does not necessarily result into an improvement for the global behavior of the schedule. Thus, we compare the performance of these strategies on practical metrics, namely the obtained efficiency and the number of running instances.

5.3. *Scheduling efficiency*

Figure 3(a) shows the efficiency obtained by each algorithm on a given scenario and for different batch sizes. The efficiency is the ratio of the obtained makespan over a lower bound computed with the steady-state algorithm (considering there is no initialization and clean-up phases) [6]. Note that we choose to present a complex example where the MIP strategy is unable to compute the optimal dependency scheme, since it is more representative of what happens in the general case. We point out that the list-scheduling heuristic has a constant behavior, as soon as the size of the batch exceeds a few tens, whereas the performance of the steady-state strategies evolves with this size: the more task graphs to schedule, the more efficient these strategies. With a very large size of batch, these strategies would all reach an efficiency of 100%, i.e., they would reach the optimal steady-state performance. In this study, we focus on batches of medium size, with a few hundreds of task graphs. We see that as soon as the batch size is larger than 200, our optimizations are able to improve the efficiency of the steady-state schedule.

We now consider the comprehensive results of all simulations. Figures 4(a) and 4(b) display the proportion of scenarios where the algorithms reach an efficiency of 90% depending on the size of the batch, both in the SIMPLE and GENERAL cases. We notice that the list-scheduling algorithm behavior does not depend on the batch size, and reaches a good performance (efficiency of 90%) only for 55% of the cases (in general). On the contrary, steady-state strategies give much better performance, reaching a good efficiency



(a) Efficiency of all algorithms on a given scenario with different batch sizes.

(b) Evolution of the number of running jobs.

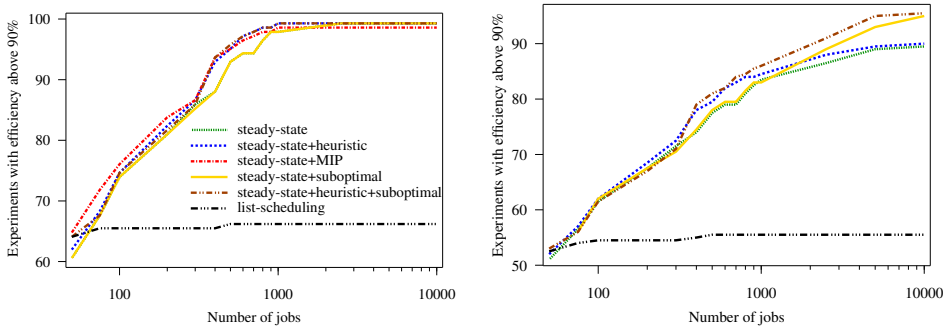
Figure 3. Examples of results for efficiency and number of running instances. (The legend is the same for both graphs.)

in 70% of the cases for batches with more than 300 task graphs. When comparing the performance of the different steady-state strategies, we notice that both optimizations help to improve the efficiency, and that their combination (with steady-state+heuristic+suboptimal) reaches the best efficiency most of the time. The gap between the optimized version of the steady-state scheduling and the original implementation is noticeable, but is not large: usually between 5% and 10% of the optimal efficiency. In the SIMPLE cases, (Figure 4(a)), we are able to compare steady-state+MIP with steady-state+heuristic: although the MIP strategy always gives better results, the heuristic performs very well, and the gap between all strategies is not always noticeable. We can also note that in these SIMPLE cases, the non-conservative period reduction is unable to improve the efficiency. Actually, SIMPLE cases correspond to scenarios where the period computed by the original steady-state approach is already short: this is why the MIP strategy is able to optimize the dependencies (the shorter the period, the smallest the optimization problem), and also why there is not much slack for period reduction. On the contrary, in GENERAL cases, the period reduction optimization really improves the efficiency.

5.4. *Buffer sizes and latency*

We have seen that our optimizations allow to slightly improve the efficiency of the steady-state approach. We now compare the schedules obtained through all variants on other metrics outlined to be important for practical use of steady-state scheduling, namely the number of task graphs (or *jobs*) being processed (which dictates the size of the buffers) and the latency (processing time of a single job). Note that both metrics are linked, since in steady-state, the total size of buffers is roughly equal to the latency times the throughput. Thus, decreasing the latency naturally helps to reduce the size of the buffers.

Figures 3(b) presents the evolution of the number of running jobs on a given platform/application scenario. At a given time t, we plot the number of jobs which have been started (some tasks have been processed), but are not terminated. Thus, temporary data



(a) Cumulated efficiency (SIMPLE) (b) Cumulated efficiency (GENERAL)

Figure 4. Cumulated efficiency. (The legend is the same for both graphs.)

for these jobs have to be stored in some buffers of the platform. This figure illustrates the typical behavior of the steady-state algorithms. During the initialization phase, the number of running jobs grows: jobs are started to prepare the next phase. During the steady-state phase, the number of running jobs is roughly constant. Finally, in the termination phase, jobs which remains in the system are terminated and the number of running jobs drops to zero. One of the drawbacks of steady-state scheduling presented in Section 2.3 is illustrated here: compared to another approach like list-scheduling, it induces a large number of running jobs (on this example, 548 instead of 3 for the list-scheduling). This example also shows that our optimizations and their combination are able to reduce the maximum number of running jobs (down to 126 jobs for the combination of our optimizations).

	SIMPLE scenarios			GENERAL scenarios		
Algorithm	average latency	maximum latency	max. num. of running jobs	average latency	maximum latency	max. num. of running jobs
MIP	94%	67%	70%	N/A	N/A	N/A
heuristic	95%	74%	76%	90%	90%	93%
suboptimal	100%	100%	100%	53%	93%	88%
heuristic+ suboptimal	95%	74%	75%	33%	67%	63%

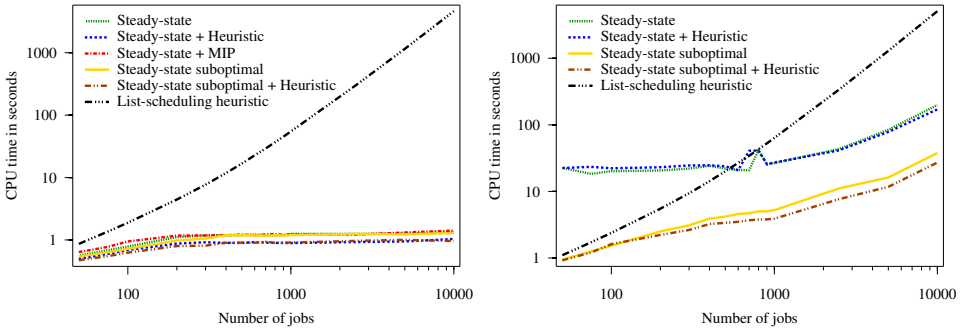
Table 1 presents comprehensive results on the average latency, the maximum latency, and the maximum number of running jobs of all our optimizations compared to the original steady-state algorithm. In the SIMPLE cases, the latency optimization is not very significant (there is even no improvement for the non-conservative period reduction), because the period is already short in these cases. On the contrary, for all cases, our optimization allows to significantly reduce the latency. For example, combining our optimizations reduces the average latency by a factor 3 (33% of the original latency), and reduces the maximum number of running jobs to 63% of the original value. Note that GENERAL cases include all cases, including the ones which are denoted SIMPLE and already have a short period.

Thus, the decrease in latency and buffer size is really important for other (complex) cases.

5.5. *Running time of the scheduling algorithms*

Figures 5(a) and 5(b) present the average time needed to compute the schedule of a batch depending on its size, in the SIMPLE and GENERAL cases. We first notice that the list-scheduling heuristic is extremely costly when the size of the batch is above a few hundreds.

In the SIMPLE cases, the time needed to optimally solve the inter-period dependency minimization using the MIP is negligible, and the time needed to compute the periodic schedule is always below 2 seconds for all strategies. In the GENERAL cases, the period of the schedule is larger, and it induces more computation: initialization and termination phases are longer (and may increase with the size of the batch), thus the computation of their schedule takes some time. The optimization of the steady-state phase by the heuristic is also time-consuming. Anyway, the computation of the schedule with steady-state approaches never exceeds 200 seconds, for a number of task graphs up to 10 000. Note that without the non-conservative period reduction, the size of the period may be very large, and the construction of the full periodic schedule is quite long (around 20 seconds on average) because of the large size of the data structures handled by the algorithms.



(a) Scheduling time (SIMPLE)

(b) Scheduling time (GENERAL)

Figure 5. Scheduling time in seconds

6. Conclusion

In this study, we have presented an adaptation of steady-state scheduling techniques for scheduling batches of task graphs in practical conditions. We consider medium-size batches (a few hundreds of jobs), and concentrate both on performance metrics (makespan) and on practical interest of the produced schedules (latencies and buffer sizes). We consider two optimizations: a better use of the periodic schedule produced by the steady-state approach through dependency reorganization, and a reduction of the periodic schedule when we allow a small throughput degradation. For the first optimization, we prove the problem

NP-complete, which motivates the design of a solution based on Mixed Integer Programming, and another heuristic solution. For the second optimization, we propose an algorithm which shorten the period while limiting the decrease of throughput to 15%. To measure the impact of our optimizations, we perform simulations, and show that both our optimizations (and their combination) improve the efficiency on medium-size batches. Furthermore, the proposed solution produces schedules that are much more practical (with smaller latency and buffer sizes). This proves that steady-state scheduling is an efficient tool for dealing with collections of task graphs, even if its analysis and optimization is a complex duty. In future works, we plan to concentrate on the tolerance of steady-state scheduling techniques to variations of the processing and communication capabilities of the platform as it may arise on a real platform.

Bibliography

[1] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Steady-state scheduling on heterogeneous clusters. *Int. J. of Foundations of Computer Science*, 16(2):163–194, 2005.

[2] D. Bertsimas and D. Gamarnik. Asymptotically optimal algorithms for job shop scheduling and packet routing. *J. Algorithms*, 33(2):296–318, 1999.

[3] H. Casanova, A. Legrand, and M. Quinson. SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In *10th IEEE International Conference on Computer Modeling and Simulation*, Mar. 2008.

[4] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23, Issue 3:187–200, July 2000.

[5] ILOG CPLEX: High-performance software for mathematical programming and optimization. http://www.ilog.com/products/cplex/.

[6] S. Diakité, J.-M. Nicod, and L. Philippe. Comparison of batch scheduling for identical multi-tasks jobs on heterogeneous platforms. In *PDP*, pages 374–378, 2008.

[7] S. Diakité, L. Marchal, J.-M. Nicod, and L. Philippe. Steady-state for batches of identical task graphs. Research report, LIP, ENS Lyon, France, Jan. 2009. available at http://graal.ens-lyon.fr/~lmarchal/pub/reports/RR2009-batches-steady-state.pdf.

[8] I. T. Foster and C. Kesselman, editors. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 2004.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[10] C. Germain, V. Breton, P. Clarysse, Y. Gaudeau, T. Glatard, E. Jeannot, Y. Legré, C. Loomis, I. Magnin, J. Montagnat, J.-M. Moureaux, A. Osorio, X. Pennec, and R. Texier. Grid-enabling medical image analysis. *Journal of Clinical Monitoring and Computing*, 19(4-5):339–349, Oct. 2005.

[11] S. Lee, M.-K. Cho, J.-W. Jung, and J.-H. K. nd Weontae Lee. Exploring protein fold space by secondary structure prediction using data distribution method on grid platform. *Bioinformatics*, 20(18):3500–3507, 2004.

[12] S. J. Ludtke, P. R. Baldwin, and W. Chiu. EMAN: Semiautomated Software for High-Resolution Single-Particle Reconstructions. *Journal of Structural Biology*, 128:82–97, 1999.

[13] T. M. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, R. M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

[14] L. Peng, K. S. Candan, C. Mayer, K. S. Chatha, and K. D. Ryu. Optimization of media pro-

cessing workflows with adaptive operator behaviors. In *Multimedia Tools and Applications*, volume 33 of *Computer Science*, pages 245–272. Springer, June 2007.

[15] J. Pitt-Francis, A. Garny, and D. Gavaghan. Enabling computer models of the heart for high-performance computers and the grid. *Philosophical Transactions of the Royal Society A*, 364(1843):1501–1516, June 2006.

[16] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Task scheduling algorithms for heterogeneous processors. In *Proceedings of HCW '99*, page 3, Washington, DC, USA, 1999. IEEE CS.