HAL
archives-ouvertes.fr

Steady-State for Batches of Identical Task Graphs

Sékou Diakité, Loris Marchal, Jean-Marc Nicod, Laurent Philippe

▶ **To cite this version:**

Sékou Diakité, Loris Marchal, Jean-Marc Nicod, Laurent Philippe. Steady-State for Batches of Identical Task Graphs. 2009. ensl-00412953

HAL Id: ensl-00412953
https://hal-ens-lyon.archives-ouvertes.fr/ensl-00412953

Submitted on 2 Sep 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Steady-State for Batches of Identical 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

LIP researh report RR2009-18

Abstract

In this paper, we focus on the problem of scheduling batches of identical task graphs on a heterogeneous platform, when the task graph consists in a tree. We rely on steady-state scheduling, and aim at reaching the optimal throughput of the system. Contrarily to previous studies, we concentrate upon the scheduling of batches of limited size. We try to reduce the processing time of each instance, thus making steady-state scheduling applicable to smaller batches. The problem is proven NP-complete, and a mixed integer program is presented to solve it. Then, different solutions, using steady-state scheduling or not, are evaluated through comprehensive simulations.

1 Introduction

Computing Grids gather large-scale distributed and heterogeneous resources, and make them available to large communities of users [7]. 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 [12], medical image processing [9], 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, which are usually materialized by files: a task produces a file which is necessary for the processing of some other task. In this paper we consider *Grid jobs* 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 [10] or independent parameter sets [14]. Moreover, the targeted applications 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 in chains or in-trees. This corresponds to application like medical [11] or media [13] image processing workflows.

The problem consists in finding a schedule for these task trees which minimizes the overall processing time, or makespan. This problem is known to be NP-hard. To overcome this issue, some of us proposed to use steady-state scheduling [1]. In steady-state scheduling, we assume that instances to be performed are 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*, we are able to derive a periodic schedule that maximizes the *throughput* of the system, that is the number of task graph instances 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. However, when the number of instances is important but bounded, existing steady-state approaches do not give optimal performances – 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 this paper, we propose an adaptation of the steady-state scheduling that allows to use it on batches of jobs of finite size, without compromising its asymptotically optimality.

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. In Section 3 we formalize the problem we are dealing with, and assess its complexity. In Section 4, we propose an exact solution and a greedy solution to this problem. Simulations showing its impact are reported in Section 5.

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, that is, $F_{k,l} = (T_k, T_l) \in E_A$ is the file produced by task T_k and consumed by task T_l. The dependency file $F_{k,l}$ has

size $\text{data}_{k,l}$. We use an unrelated computation model: computation task T_k needs a time $w_{i,k}$ to be entirely processed by processor P_i.

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 data from the others. This corresponds to the case when the same computation has to be performed on different input data sets.

2.2 Principle

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 present study is based on a steady-state approach for scheduling collections of identical task graphs proposed in [1]. The steady state is characterized using activities variables: α_i^k represent 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: $\beta_{i,j}^{k,l}$ represent the average number of files $F_{k,l}$ sent by P_i to P_j within one time unit in steady state.

By focusing in the steady state, we can write constraints on these activity variables, due to speed limitation of the processors and links. 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 that totally describe a valid steady-state schedule. We add the objective of maximizing the throughput, that is the overall number of DAGs processed by time unit, to get a linear program. Solving this linear program over the rational numbers allows us to compute the optimal steady-state throughput.

Then, from an optimal solution of this linear program, we construct a periodic schedule that achieves this optimal throughput. The construction of this schedule is complex, especially for handling communications, and we refer the interesting reader to [1] for a detailed description. In the solution of 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 L of all denominators of these quantities. We then multiply all quantities by L, to get a period where every quantities of tasks or files is integer. A period describes the activity of each processor (how many task of each types 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 will consider these communications 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.

Although bounded, the length L of the period may be large. 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 transfered 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

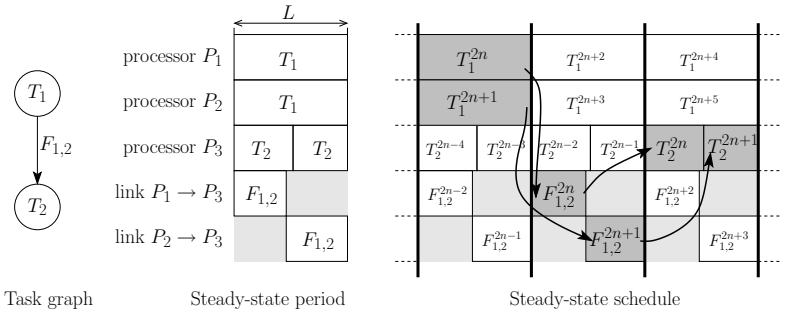


Figure 1: Handling dependencies

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 in three phases:

1. an initialization phases, where all the preliminary results needed to treat a 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.

2.3 Shortcomings

We have seen that the length 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. 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 some 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 large enough. In this study, we aim at reducing this threshold, that is to obtain a steady-state schedule which is also interesting for small batches of task graphs.

One could envision two solutions to overcome these drawbacks: (i) decrease the length of the period, or (ii) decrease the number of periods necessary to process one instance. However, there is limited hope that the first solution could be implemented if we want to reach the optimal throughput: the length of the period directly follows from the solution of the linear program. In this study, we focus on the second one, that is on reducing the latency of the processing of every instances.

3 Problem formulation and complexity

3.1 Motivation

We aim at scheduling identical DAGs (in-trees) on an heterogeneous platform with unrelated machines. Our typical workload consists in 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 the advantage of steady-state unprofitable. The overall metric is the time needed to process all the DAGs (total *makespan*). By using steady-state scheduling, we focus on *throughput* maximization. It is possible to get a solution with optimal throughput [1]; our goal is to refine this solution to make it profitable for small batches of DAGs.

The solution proposed in [1] consists in a periodic schedule. The *length of the period* of this schedule is a key parameter for our objective. However, since we want to keep an optimal throughput, we do not try to reduce this length, but we prohibit any increase in the period length, which would go against our final objective.

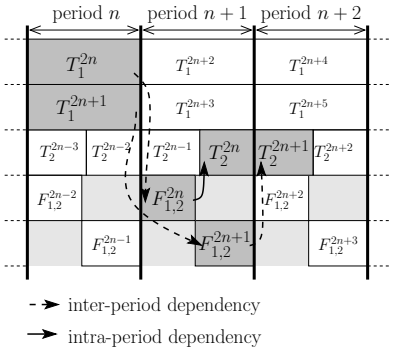


Figure 2: A periodic schedule with inter-period and intra-period dependencies

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 period, as illustrated in Figure 2. In this 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 by 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. Thus, reducing the number of such 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.

In order to get a practical implementation of steady-state scheduling for bounded sets of task graphs, we choose to forget about direct makespan minimization. We start from a period with optimal throughput (computed as in [1]), and focus on reducing the number of period dependencies in the schedule.

3.2 Formalization of the problem

We start from the description of a steady-state period. A period consists in 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 follows the edges of the task graph: for each edge $T_k \rightarrow T_l \in E_A$, for all $u = 1, \ldots, q$, we have a dependency $T_k^u \rightarrow T_l^u$.

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:

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

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

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

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 1 (INTER-PERIOD-DEP)**.** *Given a period described by σ, consisting in 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 $t(T_k^u)$ for each task instance such that the resultant number of inter-period dependencies is not larger than B?*

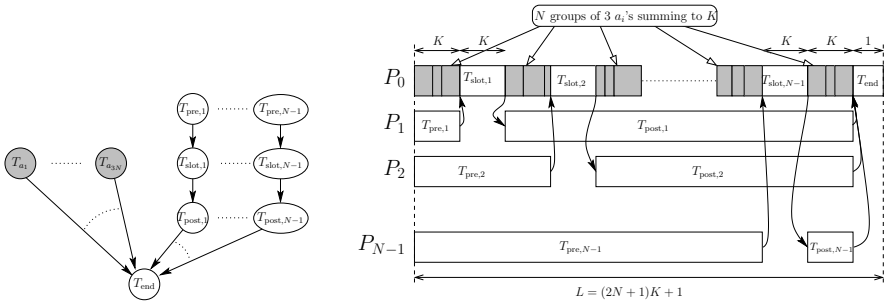
It turns out that this problem is NP-complete, as expressed by the following result.

Theorem 1. *INTER-PERIOD-DEP is NP-complete in the strong sense.*

Proof.

- We first prove that this problem belongs to NP. Once the starting times given with the function t, we can check in polynomial time that the validity constraint (1) is satisfied for any pair of tasks scheduled on the same processor, and that the number of dependencies that do not satisfy criterion (2) is upper bounded by B.
- To prove that the problem is NP-complete, we perform a reduction from the 3-PARTITION problem, which is known to be NP-complete [8]:
  Definition 2 (3-partition)**.** *Given 3n integers $a_1, \ldots, a_{3N}$, such that $\sum ai = N \times K$ and $K/4 < a_i < K/2$, is there a partition of the a_i's into N groups of 3 elements, such that each a_i belongs exactly to one group, and each group sums to K?*
  From an instance I_1 of 3-partition, we build the following instance I_2 of INTER-PERIOD-DEP:
  - The bound B is set to zero;
  - The period length is $L = (2N + 1)K + 1$;
  - The platform is made of $p = N + 1$ processors $P_0, \ldots, P_N$;
  - The task graph G_A is made of $n = 6N - 3$ tasks:
    - A task T_{end};
    - For each variable a_i $(1 \leq i \leq 3N)$, a task T_{a_i} and an edge $T_{a_i} \rightarrow T_{\text{end}}$;
    - For $1 \leq j \leq N - 1$, three tasks $T_{\text{pre},j}$, $T_{\text{slot},j}$ and $T_{\text{post},j}$, and the following edges: $T_{\text{pre},j} \rightarrow T_{\text{slot},j} \rightarrow T_{\text{post},j} \rightarrow T_{\text{end}}$.

  Figure 3(a) shows the task graph. There is only $q = 1$ instance of the task graph in the period, thus we forget about instance indices in the following. Tasks T_{a_i} $(1 \leq i \leq 3N)$, tasks $T_{\text{slot},j}$ $(1 \leq j \leq N)$ and task T_{end} are allocated to P_0; the processing times of these tasks are the following: $w_{a_i} = a_i$, $w_{\text{slot},j} = K$ and $w_{\text{end}} = 1$.
  Task $T_{\text{pre},j}$ and $T_{\text{post},j}$ are allocated to processor P_j, with processing times: $w_{pre} = (2j - 1)K$ and $w_{post} = (2N - 2j - 1)K$.



(a) Task graph

(b) Period of the schedule.

Figure 3: Reduction from 3-partition.

Note that the size of I_2 is polynomial in the size of I_1. If there exists a 3-partition of the a_i's of instance I_1, then Figure 3(b) shows how to organize the tasks so that all edges result in intra-period dependencies. This is a valid solution instance I_2.
Conversely, assume that there we are given a valid solution of instance I_2, without any inter-period dependencies. Since all dependencies of each chain $T_{\text{pre},j} \to T_{\text{slot},j} \to T_{\text{post},j} \to T_{\text{end}}$ must be satisfied during the period, all tasks $T_{\text{slot},j}$ must start at times $(2j-1)K$ as in Figure 3(b). This leaves room for N intervals of duration K during which all tasks T_{a_i} must be performed. Thus, we can extract a 3-partition of the a_i's, and exhibit a solution for I_1.

□

4 Proposed solutions

In this section, we present two solutions for the problem presented in Section 3. The first solution uses a linear program approach that makes use of both integer and rational variables, hence it is a Mixed Integer Program. Solving a MIP is NP-complete, however efficient solvers exists for this problem [5], which makes it possible to solve small instances. The second solution is a greedy algorithm that solves all possibles intra-period dependencies for a fixed period (the algorithm is not allowed to move tasks).

4.1 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 also check that a given dependency is an intra-period dependency if and only if $e_{j,k} = 1$:

$$\forall T_j \to T_k, \quad t_k - (t_j + w_j) \geq (e_{j,k} - 1) \times L \tag{5}$$

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

$$\forall P_i, \forall T_j, T_k \in \sigma(P_i), T_j \neq T_k, \quad t_k - (t_j + w_j) \geq (y_{j,k} - 1) \times L \tag{6}$$

- Finally, we check that all tasks are processed within the period:

$$\forall T_j, \quad t_j + w_j \leq L \tag{7}$$

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:

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

The previous linear program provides an optimal schedule, as expressed by the following result:

Theorem 2. *Linear program* (8) *computes a valid schedule with maximum number of dependencies satisfied within the period.*

Proof.

- We first consider a solution y, t, e of the linear program (8), and prove that it describes a valid schedule. First, note that the schedule is totally defined by the t variables, with t_j being the starting time of task T_j. Thanks to Constraint (7), all tasks are totally processed during the period. We have to check that the processing of two tasks on the same resource do not overlap. By contradiction, assume that there exists T_j and T_k in $\sigma(P_i)$ such that $t_j \leq t_k < t_j + w_j$ (T_k starts while T_k is being processed). As $t_j \leq t_k$ and both tasks are executed within the period, we have $-T < t_j - t_k < 0$. According to Constraint (3), we thus have $y_{j,k} = 1$. Thanks to Constraints (6), we then have $t_k - (t_j + w_j) \geq 0$, which contradicts $t_k < t_j + w_j$. Thus, two tasks processed on the same resource cannot overlap. At last, the number of intra-period dependencies in the schedule is exactly D, thanks to Constraint (5).

- We now consider the value D_{opt} of the objective in an optimal solution. We prove that D_{opt} is an upper bound on the number of intra-period dependencies for any valid schedule. From any schedule, we can easily

construct y, e and t variables that describe the schedule as explained above. These variables must verify Constraints (3) and (4) because of the definition of y and t. They also verify Constraints (5) by definition of the e's. Since the schedule is valid, no task $T_k \neq T_j$ is processed during interval $[t_j, t_j + w_j]$, thus Constraints (6) hold. Finally, all tasks are processed within a period, thus Constraints (7) are verified. Thus, y, t, e is a solution of the linear program, and its objective value D is lower than the optimal value D_{opt}. □

4.2 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 types 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 occurrences 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 try 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.

When we have applied the previous greedy algorithm on all possible dependency types $T_k \rightarrow T_l$, then each task is part of a complete task graph. Note that this task graph corresponds to the original one since we are targeting only tree-shaped graph (either in-trees or out-trees). Then we simply have to annotate the different task graphs with the original instance indices in order to get a complete valid schedule.

5 Experimental results

In this Section, we present experimental results that show how minimizing the inter-period dependencies improves the original steady-state algorithm. We compare four algorithms that schedule batches of identical jobs on a heterogeneous platform. The first algorithm is the original steady-state implementation and the second algorithm the steady-state implementation with the optimization using mixed integer programming to minimize the number of inter-period dependencies described above (called steady-state+MIP). The third algorithm is also a steady-state implementation with inter-period dependencies minimization, but we replace the MIP optimization with a simple greedy algorithm, called steady-state+heuristic. The fourth algorithm is a classical list-scheduling algorithm based on HEFT [15]: 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 (EFT). The EFT evaluation depends on the load of the platform and takes both the computation time and the communication time into account. Note that in steady-state strategies, the initialization and clean-up phases are implemented using this list-scheduling technique.

5.1 Simulation settings

In the following, we report simulation results obtained with a simulator implemented above SimGrid and its MSG API [3]. The experiences consist in the simulation of 245 platform/application scenarios for batches from 1 to 1000 jobs. The platforms are randomly generated; parameters allows platforms from 4 to 10 nodes. Each nodes can process a subset of the 10 different task types with a computing cost between 1 and 11 time units. Network links generation ensures that the platform graph is connected, links bandwidth are homogeneous. The applications are also randomly generated, generation parameters allows in-trees with 5 to 15 nodes. Nodes types are selected between the 10 different task types. Dependency generation ensures that the application graph is an in-tree, dependency file sizes vary from 1 to 2.

For some of the scenarios, large periods and large numbers of dependencies may arise and the optimal dependency reduction with the MIP becomes too costly to compute, even though an efficient MIP solver is used (CPLEX [5]). In the following, we thus distinguish two cases: SIMPLE scenarios are the ones when we are able to solve the MIP (139 scenarios), and GENERAL scenarios gathers all cases, and we do not include MIP results (245 scenarios).

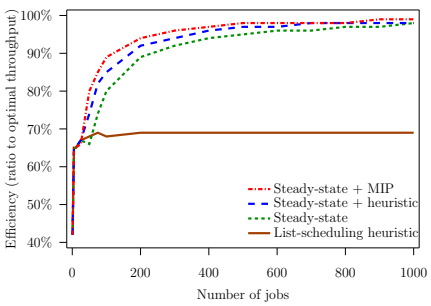
5.2 Number of inter-period dependencies

In the simulations, we count 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 39% of the inter-period dependencies, whereas the heuristic is able to suppress only 29% to 30% of them (29% in all cases, and 30% in SIMPLE cases). This shows that both the MIP and the heuristic strategies achieve a good performance for our metric. As we have outlined in the introduction, this does not necessarily result into an improvement for the global behavior of the schedule. Thus, we

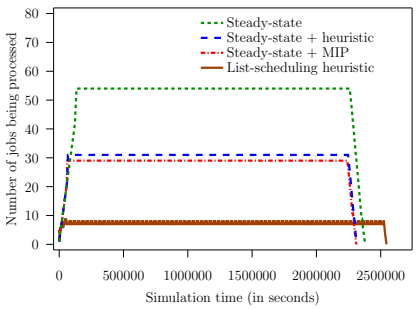
also compare the performance of these strategies on other metrics, namely the obtained throughput and the number of running instances.

5.3 Scheduling efficiency

Figure 4(a) shows the performance of the four scheduling algorithms on a given scenario. The efficiency, that is the ratio of the optimal throughput, obtained by each algorithm is given for different batch sizes. 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 jobs 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 give the optimal steady-state throughput. In this study, we focus on batches with medium size. On this particular example, all steady-state strategies achieve 90% of the optimal throughput as soon as there are 300 jobs to schedule.



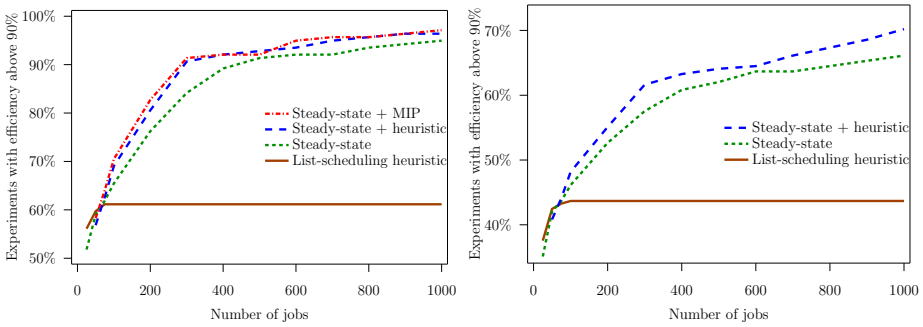
(a) Efficiency for batches of increasing size.



(b) Evolution of the number of running instances.

Figure 4: Examples of results for efficiency and number of running instances.

Figures 5(a) and 5(b) display the proportion of scenarios where the algorithms reach 90% of the optimal throughput, 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 (90% of the optimal throughput) only for 43% of the cases (in general). On the contrary, steady-state strategies give much better performance, reaching a good throughput in 60% of the cases for batches with more than 400 jobs. Here, we are interested in comparing the performance of the different steady-state strategies. We notice that the performance of steady-state+MIP and steady-state+heuristic is better than steady-state: for medium-size batches, the number of jobs needed to get a good performance is smaller than in the original steady-state algorithm. This gap is noticeable even if it is not very large. In the SIMPLE case, (Figure 5(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 both strategies is not always noticeable.



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

Figure 5: Cumulated efficiency

5.4 Number of running instances

Figures 4(b) presents the evolution of the number of running job instances on a given platform/application scenario. At a given time t, a running instance is a job which has been started (some tasks have been processed), but is not terminated at time t. Thus, temporary data for this instance 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 job instances grows: instances are started to prepare the next phase. During the steady-state phase, the number of job instances is roughly constant. Finally, in the termination phase, remaining instances are processed and the number of running instances drops to zero.

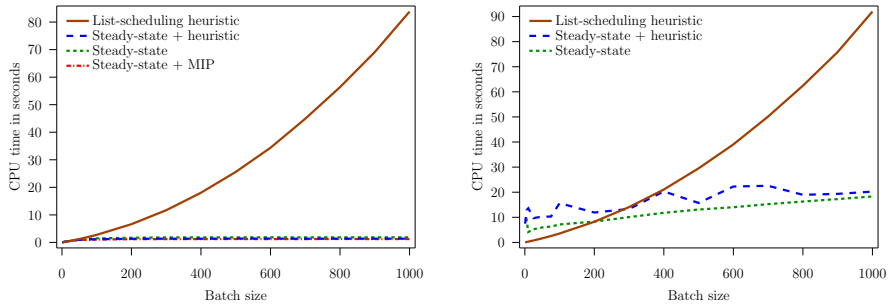
One of the drawbacks of steady-state scheduling presented in Section 2.3 is illustrated here: compared to another approach like list-scheduling, it induces a large number of running instances (on this example, 54 instead of 8). This example also shows that reducing the number of inter-period dependencies reduces the number of running instances for steady-state scheduling: with the MIP optimization, we get a maximum of 29 running instances, and 31 with the heuristic. We compared the maximum number of running instances in steady-state for the optimized versions (MIP and heuristic) and the original one: on average, steady-state+MIP induces a decrease of 35% and steady-state+heuristic reaches a decrease between 22% and 26% (respectively in the GENERAL and SIMPLE cases). Thus, our optimization makes steady-state scheduling more practical as it reduces the size of the required buffers.

5.5 Running time of the scheduling algorithms

Figures 6(a) and 6(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. Its supra-linear behavior is due to the complexity of finding a ready task to schedule in a number of considered tasks that grows linearly in the size of the batch.

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 20 seconds.



(a) Scheduling time (SIMPLE)

(b) Scheduling time (GENERAL)

Figure 6: 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, that is when the size of the batch is limited. The optimization we propose consists in a better usage of the period of the steady-state schedule. Instead of directly targeting the minimization of the makespan, we choose to reduce the number of inter-period dependencies. This problem is NP-complete, which justifies a solution based on Mixed Integer Programming. Our simulations show that this objective was relevant: when decreasing the number of inter-period dependencies, the throughput of the solution on medium-size batches is improved. Furthermore, the obtained solution requires less buffer space, since fewer instances are processed simultaneously, making the schedule even more practical. In future work, we plan to concentrate on small-size batches: since the optimal throughput is not reachable for these batches, it would be interesting to study non-conservative approaches, i.e., periodic schedules based on sub-optimal throughput, but which are more convenient to use thanks to their short period.

References

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] L. Peng, K. S. Candan, C. Mayer, K. S. Chatha, and K. D. Ryu. Optimization of media processing workflows with adaptive operator behaviors. In *Multimedia Tools and Applications*, volume 33 of *Computer Science*, pages 245–272. Springer, June 2007.

[14] 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.

[15] 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.