INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Allocating Series of Workflows on Computing Grids

Matthieu Gallet — Loris Marchal — Frédéric Vivien

N° 6603

July 2008

Thème NUM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--6603--FR+ENG

Allocating Series of Workflows on Computing Grids

Matthieu Gallet , Loris Marchal, Frédéric Vivien

Thème NUM — Systèmes numériques
Projet GRAAL

Rapport de recherche n° 6603 — July 2008 — 28 pages

Abstract: In this report, we focus on scheduling jobs on computing Grids. In our model, a Grid job is made of a large collection of input data sets, which must all be processed by the same task graph or *workflow*, thus resulting in a *series of workflow* problem. We are looking for an efficient solution with regard to throughput and latency, while avoiding solutions requiring complex control. We thus only consider single-allocation strategies. We present an algorithm based on mixed linear programming to find an optimal allocation, and this for different routing policies depending on how much latitude we have on communications. Then, using simulations, we compare our allocations to reference heuristics. Our results show that our algorithm almost always finds an allocation with good throughput and low latency, and that it outperforms the reference heuristics, especially under communication-intensive scenarios.

Key-words: Workflows, DAGs, scheduling, heterogeneity, computing Grid.

This text is also available as a research report of the Laboratoire de l'Informatique du Parallélisme `http://www.ens-lyon.fr/LIP`.

Allocation d'une série de graphes de tâches sur une grille de calcul.

Résumé : Dans ce rapport, nous nous intéressons à l'ordonnancement d'applications de type *workflow* sur une grille de calcul. Dans notre modèle, une telle application est formée d'une grande collection de données qui doivent toutes être traitées avec le même graphe de tâche. Nous avons donc une séries de graphes de tâches à effectuer. Nous recherchons une solution efficace pour le débit et la latence, tout en nous interdisant le recours à un contrôle trop complexe. C'est pourquoi nous nous concentrons sur les solutions utilisant une seule allocation. Nous présentons un algorithme utilisant la programmation linéaire mixte qui calcule une allocation de débit optimal, et ce pour différentes politiques de routage, en fonction de la latitude que nous avons sur les communications. Puis, grâce à des simulations, nous comparons ces allocations à des heuristiques de référence. Nos résultats montrent que notre algorithme est toujours capable de trouver une allocation de bon débit et de latence limitée et qu'il surpasse les résultats des heuristiques de référence, en particulier lorsque les temps communications sont prépondérants.

Mots-clés : Workflows, graphes de tâches, ordonnancement, hétérogénéité, grille de calcul.

1 Introduction

Computing Grids gather large-scale distributed and heterogeneous resources, and make them available to large communities of users [18]. Such Grids 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 [11], bioinformatics [29], medical image processing [22], etc. Distributing an application on such a platform is an increasingly 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. In this paper, we investigate the problem of mapping an application onto the computing platform. We are both interesting in optimizing the performance of the mapping (that is, process the data as fast as possible), and to keep the deployment simple, so that we do not have to deploy complex control softwares on a large number of machines.

Applications are usually described by a (directed) graph of tasks, what is usually called a *workflow* in the Grid literature. The nodes of this graph represent the computing tasks, while the edges between nodes stand for the dependences 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* involving a large collection of input data sets that must all be processed by the same application. In other words, the *Grid jobs* we consider are made of the same workflow applied to a large collection of different input data sets. We can evenly consider that we have a large number of instances of the same task graph to schedule. Such a situation arises when the same computation must be performed on independent data [28], independent parameter sets [32], or independent models [26]. A classical example lies in image processing: the data set is a large number of input images, and all images have to be processed the same way, for example by using several consecutive filters and encoders.

In this paper, we concentrate on how to map several instances of a same workflow onto a computing platform, that is, on how to decide on which resource a task has to be processed, and on which route a file has to be transfered, if we assume that we have some control on the routing mechanism. We will study several scenarios, with different routing policies.

We start by motivating our problem (Section 2). Then we formally define our problem (Section 4) and describe our solution (Section 5). Finally, we assess the quality of these solutions through simulations (Section 6) and conclude (Section 7).

2 Problem motivation

In this section we motivate the application model we work with.

2.1 Dynamic scheduling vs. static scheduling

Many scheduling strategies use a *dynamic* approach: task graphs, or even tasks, are processed one after the other. This is usually done by assigning priorities to waiting tasks, and then by allocating resources to the task with highest priority, as long as there are free resources. This simple strategy is the best possible in some cases: (i) when we have very little knowledge on the future workload (i.e., the tasks that will be submitted in the near future, or released by the processing of current tasks), or (ii) under a very unstable environment, where machines join and leave the system with a high churn rate. However, in the case of scheduling a series of workflows, the number of workflows is assumed to be large, and the number of resulting tasks is even larger. If a dynamic scheduler is used to schedule this large number of tasks, it would result in a large processing time by the scheduler, or even worse, in an overflow in the task buffer. This way, the scheduler would be unavailable for other users until all our tasks are scheduled, which would be unfair in a multi-user environment.

On the contrary to the typical use case of dynamic schedulers, we have much knowledge on the system when scheduling several instances of a workflow. First, we can take advantage of the regularity of the workflows: the input is made of a large collection of data sets that will result in the same task graph. Second, the computing platform is considered to be stable enough so that we can use performance measurement tools like NWS [37] in order to get some information on machine speeds and link bandwidths. Taking advantage of this knowledge, we aim at using *static* scheduling techniques, that is to anticipate the mapping and the scheduling of the whole workload at its submission date.

2.2 Data parallelism vs. control parallelism

In the context of scheduling series of task graphs, we can take advantage of two sources of parallelism to increase performance. First, parallelism comes from the *data*, as we have to process a large number of instances. Second, each instance consists in a task graph which may well include some parallelism: some tasks can be processed simultaneously, or the processing of consecutive tasks of different instances can be pipelined, using some *control* parallelism. In such a context, several scheduling strategies may be used.

We may only make use of the data parallelism. In this case, the whole workflow corresponding to the processing of a single input data set is executed on a single resource, as if it was a large sequential task. Different workflow instances are simultaneously processed on different processors. This is potentially the solution with the best degree of parallelism, because it may well use all available resources. This imposes that all tasks of a given instance are performed on each processor, therefore that all services must be available on each participating machine. However, it is likely that some services have heterogeneous performance: many legacy codes are specialized for specific architectures and would perform very poorly if run on other machines. Some services are even likely to be unavailable on some machines. In the extreme, most specified case, it may happen that no machine can run all services; in that case the pure data-parallelism approach is infeasible. Moreover, switching, on the same machine, from one service to another may well induce some latency to deploy each service, thus leading to a large overhead. At last, a single input data set may well produce a large set of data to process or require a large amount of memory. Processing the whole workflow on a single machine may lead to a large latency for this instance, and may even not be possible if the available storage capacity or memory of the machine cannot cope with the workflow requirements. For these reasons, application workflows are usually not handled using a pure data-parallelism approach.

Another approach consists in simultaneously taking advantage of both data and control parallelism. We have previously studied this approach [5, 6] and proved that in a large number of cases, when the application graph is not too deep, we can compute an optimal schedule, that is a schedule which maximizes the system throughput. This approach, however, asks for a lot of control as similar files produced by different data sets must follow different paths in the interconnection network.

In this paper, we focus on a simpler framework: we aim at finding a single mapping of the application workflow onto the platform, with good performance. This means that all instances of a given task must be processed on the same machine. Thus, the corresponding service has to be deployed on a single machine, and all instances are processed the same way. Thus, the control of the Grid job is much simpler, and the number of needed resources is kept low.

2.3 Steady-state operation and throughput maximization

As in our previous work for scheduling application graphs on heterogeneous platforms, this study relies on *steady-state* scheduling. The goal is to take advantage of the regularity of the series of workflows; as we consider that the Grid job input is made of a large number of data sets which should be processed using the same task graph, we relax the scheduling problem, and consider the *steady-state* operation: we assume that after some transient initialization phase, the throughput of each resource will become steady.

In scheduling, the classical objective is to minimize the running time of the job, or *makespan*. However, by using steady-state techniques, we relax this objective and concentrate on maximizing the system throughput. Then, the total running time is composed of one initialization phase, a steady-state phase, and a clean-up phase. As initialization and clean-up phases do not depend on the total number of instances, we end up with asymptotically optimal schedules: when the number of instances becomes large, the time needed to perform initialization and clean-up phases becomes negligible in front of the overall running time.

3 Related work

In this section, we report the related work link to workflow scheduling and steady-state relaxation.

Workflows. Managing and scheduling workflow on computing Grids is the subject of a wide literature. Many middlewares are develop in order to manage workflows on the Grid. One of the most comprehensive, as far as scheduling is concerned in probably MOTEUR [23]. All these tools usually include scheduling heuristics to map the tasks of the workflow onto the available resources. These heuristics are inherited from DAG scheduling, and more or less adapted to Grid environment to cope with its intrinsic heterogeneity. Among other, these techniques make use of list scheduling heuristics [12], clustering [30] and task duplication [2]. Many meta-heuristics are derived by assembling various scheduling heuristics. Note that some workflow may include parallel tasks, requiring a (fixed or not) number of processors, and some scheduling techniques take advantage of this knowledge [34]. Since workflows and Grid computing are very popular nowadays, we must be careful with the vocabulary: a workflow is most of the time considered to be a single graph of tasks with dependences, but sometimes also denotes a pipeline usage of this task graph, like our "series of workflow" setting.

Steady-state. Minimizing the makespan, i.e., the total execution time, is a NP-hard problem in most practical situations [21, 33, 12], while it turns out that the optimal steady-state schedule can often be characterized very efficiently, with low-degree polynomial complexity.

The steady-state approach has been pioneered by Bertsimas and Gamarnik [8]. It has been used successfully in many situations [7]. In particular, steady-state scheduling has been used to schedule independent tasks on heterogeneous tree-overlay networks [4, 3]. The steady-state approach has also been used by Hong et al. [27] who extend the work of [4] to deploy a divisible workload on a heterogeneous platform.

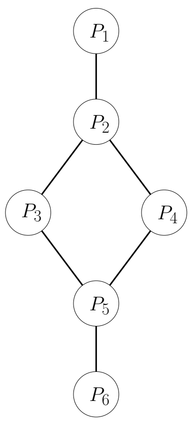
4 Notations, hypotheses, and problem complexity

4.1 Platform and application model

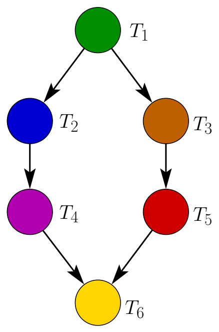
We denote by $G_P = (V_P, E_P)$ the undirected graph representing the platform, where $V_P = \{P_1, \dots, P_p\}$ is the set of all processors. The edges of E_P represent the communication links between these processors. The maximum bandwidth of the communication link $P_q \to P_r$ is denoted by $\mathrm{bw}_{q,r}$. Moreover, we suppose that processor P_q has a maximum incoming bandwidth B_q^{in} and a maximum outgoing bandwidth B_q^{out}. Figure 1(a) gives an example of such a platform graph.

We use a bidirectionnal multiport model for communications: a processor can perform several communications simultaneously. In other words, a processor can simultaneously send data to multiple targets and receive data from multiple sources, as soon as the bandwidth limitation is exceeded neither on links, nor on incoming or outgoing ports.

A path from processor P_q to processor P_r, denoted $P_q \rightsquigarrow P_r$, is a set of adjacent communication links going from P_q to P_r.



(a) Platform graph



(b) Application graph

Figure 1: Examples of platform and applications.

We denote by $G_A = (V_A, E_A)$ the Directed Acyclic application Graph (DAG), where $V_A = \{T_1, \ldots, T_n\}$ is the set of tasks, and E_A represents the dependences between these tasks, that is, $F_{i,j} = (T_i, T_j) \in E_A$ is the file produced by task T_i and consumed by task T_j. The dependence file $F_{i,j}$ has size $\mathrm{data}_{i,j}$, so that its transfer through link $P_q \to P_r$ takes a time $\frac{\mathrm{data}_{i,j}}{\mathrm{bw}_{q,r}}$. Computation task T_k needs a time $w_{i,k}$ to be entirely processed by processor P_i. Figure 1(b) gives an example of application graph. This last notation corresponds to the so-called unrelated-machines model: a processor can be fast for a given type of task and slow for another one. Using these notations, we can model the benefits which can be drawn on some specific hardware architectures by specially optimized tasks. For example, a Cholesky factorization can be 5.5 times faster when using a GeForce 8800GTX graphic card than when using only the CPU, while a LU factorization is only 3 times faster in the same conditions [36]. Even higher speed-ups can be reached on some graph applications, as high as 70 [25]. Unrelated performance may also come from memory requirements. Indeed, a given task requiring a lot of memory will be completed faster when processed by a slower processor with a larger amount of memory. Grids are often composed of several clusters bought over several years, thus with very different memory capacities, even when processors are rather similar.

Remarks on platform modeling. The way we model the platform aims at taking into account all processing units and routers in the set of nodes V_P. Modelling routers that should not be enrolled in computations is straightforward by using infinite computation time on these resources. Similarly, E_P is the set of all physical links connecting nodes of V_P. The advantage of this approach is that it allows us to correctly model potential congestions in the platform. Its main drawback is the difficulty of first acquiring a good model of the topology of the interconnection network, and then of instantiating its parameters. Although this is a tough problem, some solutions begin to arise [17].

Another way to consider the model is at an application-level. From this point of view, all nodes in V_P are processors under our control —i.e, on which we can deploy our application— and E_P is the set of logical routes connecting these nodes (each logical route can then account for several physical communication links). This model has the advantage of the simplicity: it does not need information on physical links or routers which may be out of the scope of our knowledge. On the other hand, potential link contentions are not taken into account, precisely because physical links are not directly considered.

4.2 Allocations

As described in the introduction, we assume that a large set of input data sets has to be processed. These input data sets are originally available on a given source processor P_{source}. Each of these data sets contains the data for the execution of one instance of the application workflow. For the sake of simplicity, we are looking for strategies where all tasks of a given type T_i are performed on the same resource, which means that the allocation of tasks to processors is the same for all instances. We now formally define an allocation.

Definition 1 (Allocation)**.** *An allocation of the application graph to the platform graph is a function σ which associates:*

- *to each task T_i, a processor $\sigma(T_i)$ which processes all instances of T_i;*
- *to each file $F_{i,j}$, a set of communication links $\sigma(F_{i,j})$ which carries all instances of this file from processor $\sigma(T_i)$ to processor $\sigma(T_j)$.*

A file $F_{i,j}$ may be transfered differently from $\sigma(T_i)$ to $\sigma(T_j)$ depending on the routing policy enforced on the platform. We distinguish three possible policies:

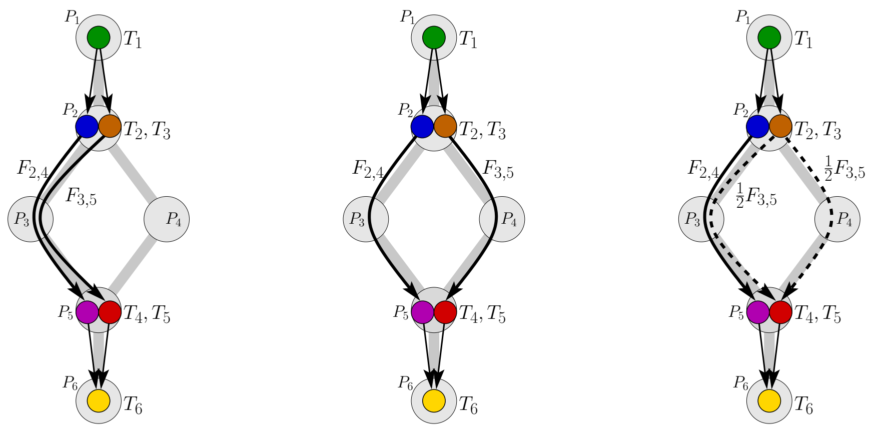
Single path, fixed routing. The path for any transfer from P_q to P_r is fixed a priori. We do not have any freedom on the routing. This scenario corresponds to the classical case where we have no freedom on the routing between machines: we cannot change the routing tables of routers.

Single path, free routing. We can choose the path from P_q to P_r, but a single route must be used for all data originating from P_q and targeting P_r. This policy corresponds to protocols allowing us to choose the route for any of the data transfer, and allowing us to reserve some bandwidth on the chosen routes. Although current network protocols do not provide this feature, bandwidth reservation, and more generally resource reservation in Grid network, is the subject of a wide literature, and will probably be available in future computing platforms [19].

Multiple paths. Data from P_q to P_r may be split along several routes taking different paths. This corresponds to the uttermost flexible case where we can simultaneously reserve several routes and bandwidth fractions for concurrent transfers.

Our three routing policies allow us to model a wide range of practical situations, current and future. The techniques exposed in Section 5 enable us to deal with any of this model and even with combinations of them.

In the case of single path policies, $\sigma(F_{i,j})$ is the set of the links constituting the path. In the case of multiple paths, $\sigma(F_{i,j})$ is a weighted set of paths $\{(w_\alpha, P_a)\}$: for example $\sigma(F_{7,8}) = \{(0.1, P_1 \rightarrow P_3), (0.9, P_1 \rightarrow P_2 \rightarrow P_3)\}$ means that 10% of the file $F_{7,8}$ go directly from P_1 to P_3 and 90% are transfered through P_2.



(a) Single path, fixed routing. (b) Single path, free routing. (c) Multiple paths.

Figure 2: Allocation examples for various routing policies.

In Figure 2, we give, for each routing policy, an example of allocation of the application graph of Figure 1(b). In all cases, the mapping of tasks is the same: $\sigma(T_1) = P_1$, $\sigma(T_2) = \sigma(T_3) = P_2$, $\sigma(T_4) = \sigma(T_5) = P_5$ and $\sigma(T_6) = P_6$. In Figure 2(a), the path for any transfer is fixed; in particular, all data from P_2 targeting P_5 must pass through P_3. In Figure 2(b), we assume a free routing policy: we can choose to route some files via P_3 and some others via P_4. At last, in Figure 2(c), we are allowed to use multiple paths to route a single file, which is done for file $F_{3,5}$.

We can also look at our routing policies from an application point of view, as we did with the platform model. Then, from this perspective:

- At an application level, the standard policy would be *single path with fixed routing*: the application is the passive subject of external routing decisions it does not try to interfere with.

- Under the *single path with free routing* policy, the application tries to improve its performance by circumventing some of the routing decisions. In practice, a communication is not made, at an application-level, directly from a participating node P_q to a participating node P_r, but may be routed through other participating nodes. The use of a single path for each transfer naturally derives from the need to have a simple control, and is in strict accordance with the use of a single allocation.

- Using several concurrent routes with the *multiple paths* policy may considerably increase the system's throughput if the application is communication intensive. However this would come at the price of more control on the participating nodes, because of the added necessity to split transfers into pieces to be separately routed.

Thus, our platform model and routing policies can be used to model a wide variety of situations, depending on how much control we are allowed, or we want, on the platform. Moreover, this is not a comprehensive list of possibilities, and we can well imagine to incorporate other policies, or a combination of them.

4.3 Throughput

We first formally define what we call the "throughput" of a schedule. Then, we will derive a tight upper bound on the throughput of any schedule.

The definition of throughput. We focus on the optimization of the steady state. Thus, we are not interested in minimizing the execution time for a given number of workflow instances, but we concentrate on maximizing the throughput of a solution, that is the average number of instances that can be processed per time-unit in steady-state.

Definition 2. *Assume that the number of instances to be processed is infinite, and note $N(t)$ the number of instances totally processed by a schedule at time t. The throughput ρ of this schedule is given by $\rho = \lim_{t\to\infty} \frac{N(t)}{t}$.*

This definition is the most general one, as it is valid for any schedule. In this study, we are interested in very specific schedules, consisting of only one allocation. We now show how to compute an upper bound on the achievable throughput of a given allocation. We will later show that this bound is tight.

Upper bound on the achievable throughput. First, we consider the time spent by each resource on one instance of a given allocation σ. In other words, we consider the time spent by each resource for processing a single copy of our workflow under allocation σ.

- The computation time spent by a processor P_q for processing a single instance is: $t_q^{\text{comp}} = \sum_{i,\sigma(T_i)=P_q} w_{i,q}$.

- The total amount of data carried by a communication link $P_q \to P_r$ for a single instance is $d_{q,r} = \sum_{(i,j), P_q\to P_r \in \sigma(F_{i,j})} \text{data}_{i,j}$ in the case of single-path policies, and it is

$d_{q,r} = \sum_{F_{i,j}} \sum_{\substack{(W_a,P_a)\in\sigma(F_{i,j}) \\ P_q\to P_r\in P_a}} w_a \times \text{data}_{i,j}$ in the case of the multiple-paths policy. This allows us to compute the time spent by each link, and each network interface, on this instance:

- on link $P_q \to P_r$: $t_{q,r} = d_{q,r}/\text{bw}_{q,r}$;
- on outgoing interface of P_q: $t_q^{\text{out}} = \sum_r d_{q,r}/B_q^{\text{out}}$;
- on incoming interface of P_q: $t_q^{\text{in}} = \sum_r d_{r,q}/B_q^{\text{in}}$.

We can now compute the maximum time τ spent by any resource for the processing of one instance: $\tau = \min\left\{\min_{P_q}\{t_q^{\text{comp}}, t_q^{\text{out}}, t_q^{\text{in}}\}, \min_{P_q\to P_r} t_{q,r}\right\}$. This obviously gives us an upper bound on the achievable throughput: $\rho \le \rho_{\max} = 1/\tau$. Indeed, as there is at least one resource which spends a time τ to process its share of a single instance, the throughput cannot be greater than 1 instance per τ units of time. We now show that this upper bound is achievable in practice, i.e., that there exists a schedule with throughput $\rho_{\max}$. In the following, we call "throughput of an allocation" the optimal throughput $\rho_{\max}$ of this allocation.

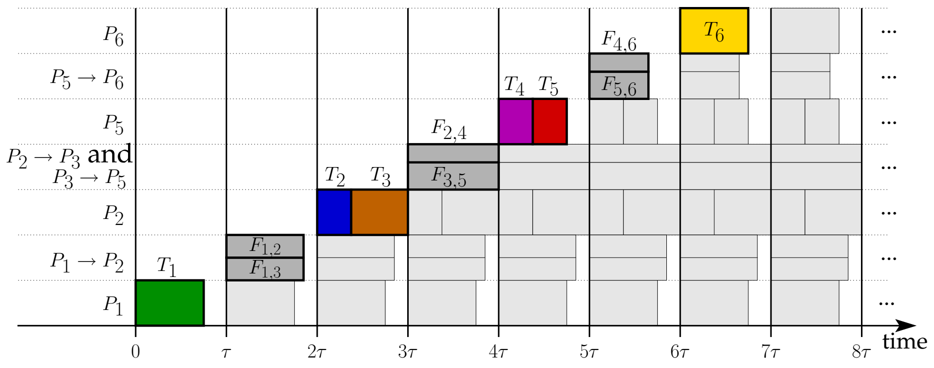


Figure 3: Example of a periodic schedule for the allocation represented on Figure 2(a). Only the first instance is represented with task and file labels.

The upper bound is achievable. Here, we will only explain on an example how one can built a periodic schedule achieving the throughput $\rho_{\max}$. Indeed, we are not interested here in giving a formal definition of periodic schedules, or to formally define and prove schedules which achieve the desired throughput, as this goes far beyond the scope of this paper. The construction of such schedules, for applications modeled by DAGs, was introduced in [5], and a fully comprehensive proof can be found in [6].

Figure 3 illustrates how to build a periodic schedule of period τ for the workflow described on Figure 1(b). Once the schedule has reached its steady-state, that is after 6τ in the example, during each period, each processor computes one instance of each task assigned to it. More precisely, in steady-state, during period k ($k \geq 6$), that is during time-interval $[k\tau; (k+1)\tau]$, the following operations happens:

- P_1 computes task T_1 of instance k,
- P_1 sends $F_{1,2}$ and $F_{1,3}$ of instance $k-1$ to P_2,
- P_2 processes T_2 and T_3 of instance $k-2$,
- P_2 sends $F_{2,4}$ and $F_{3,5}$ of instance $k-3$ to P_5 (via P_3),
- P_4 processes tasks T_4 and T_5 of instance $k-4$,
- P_5 sends $F_{4,6}$ and $F_{5,6}$ of instance $k-5$ to P_6,
- P_6 processes task T_6 of instance $k-6$.

One instance is thus completed after each period, achieving a throughput of $1/\tau$.

4.4 NP-completeness of throughput optimization

We now formally define the decision problem associated to the problem of maximizing the throughput.

Definition 3 (DAG-Single-Alloc)**.** *Given a directed acyclic application graph G_A, a platform graph G_P, and a bound B, is there an allocation with throughput $\rho \geq B$?*

Theorem 1. *DAG-Single-Alloc is NP-complete for all routing policies.*

Proof. We first have to prove that the problem belongs to NP, that is that we can check in polynomial time that the throughput of a given allocation is greater than or equal to B. Thanks to the previous section, we know that this check can be made through the evaluation of a simple formula; DAG-Single-Alloc is thus in NP.

To prove that DAG-Single-Alloc is NP-complete, we use a reduction from the Minimum Multiprocessor Scheduling, known to be NP-complete [21]. Consider an instance $\mathcal{I}_1$ of Multiprocessor Scheduling, that is a set of n independent tasks $T_{i,1\leq i\leq n}$ and a set of m processors $P_{u,1\leq u\leq m}$, where task i takes time $t(i,u)$ to be processed on processor P_u. The problem is to find a schedule with total execution time less than a given bound T. We construct a very similar instance of DAG-Single-Alloc:

- The application DAG is a simple fork, made of all tasks T_i plus a task T_0, root of the fork: for each $1 \leq i \leq n$, there is an edge $F_{0,i}$, with $\mathrm{data}_{0,i} = 0$.
- The platform consists of the same set of processors than $\mathcal{I}_1$, connected with a complete network where all bandwidths are equal to 1. The time needed to process task T_i on processor P_u is $w_{i,u} = t(i,u)$ for each $1 \leq i \leq n$, and $w_{0,u} = 0$.

Note that communications need not being taken into account during performance evaluation, since all data sizes are null. Thus, this reduction applies to any routing policy. The throughput of an allocation is directly related to the total execution time of the set of tasks: an allocation has throughput ρ if and only if it completes all the tasks in time $1/\rho$. Thus finding a schedule with completion time less than T is equivalent to finding an allocation with throughput greater than $1/T$. □

5 Mixed linear program formulation for optimal allocations

In this section, we present a mixed linear program formulation that allows to find optimal allocation with respect to the total throughput.

5.1 Single path, fixed routing

In this section, we assume that the path to be used to transfer data from a processor P_q to a processor P_r is determined in advance; we have thus no freedom on its choice. We then denote by $P_q \rightsquigarrow P_r$ the set of edges of E_P which are used by this path.

Our linear programming formulation makes use of both integer and rational variables. The resulting optimization problem, although NP-complete, is solvable by specialized softwares (see Section 6 about simulations). The integer variables can take 0 or 1 value. The only integer variables are the following:

- y's variables which characterize where each task is processed: $y_q^k = 1$ if and only if task T_k is processed on processor P_q;
- x's variables which characterize the mapping of file transfers: $x_{q,r}^{k,l} = 1$ if and only if file $F_{k,l}$ is transfered using path $P_q \rightsquigarrow P_r$; note that we may well have $x_{q,q}^{k,l} = 1$ if processor P_q executes both tasks T_k and T_l.

Obviously, these two sets of variables are related. In particular, for any allocation, $x_{q,r}^{k,l} = y_q^k \times y_r^l$. This redundancy allows us to write linear constraints.

Linear program (1) expresses the optimization problem for the fixed-routing policy. The objective function is to minimize the maximum time τ spent by all resources, in order to maximize the throughput $1/\tau$. The intuition behind the linear program is the following:

- Constraints (1a) define the domain of each variable: x, y lie in $\{0, 1\}$, while τ is rational.
- Constraint (1b) ensures that each task is processed exactly once.
- Constraint (1c) asserts that a processor can send the output file of a task only if it processes the corresponding task.
- Constraint (1d) asserts that the processor computing a task holds all necessary input data: for each predecessor task, it either received the data from that task or computed it.
- Constraint (1e) ensures that the computing time of a processor is no larger that τ.
- In Constraint (1f), we compute the amount of data carried by a given link, and the following constraints (1g,1h,1i) ensure that the time spent on each link or interface is not larger than τ, with a formulation similar to that of Section 4.3.

We denote $\rho_{\text{opt}} = 1/\tau_{\text{opt}}$, where τ_{opt} is the value of τ in any optimal solution of Linear Program (1). The following theorem states that ρ_{opt} is the maximum achievable throughput.

Theorem 2. *An optimal solution of Linear Program* (1) *describes an allocation with maximal throughput for the fixed routing policy.*

$$
(1)\quad \left\{
\begin{array}{lll}
\multicolumn{3}{l}{\text{MINIMIZE } \tau \text{ UNDER THE CONSTRAINTS}} \\
\text{(1a)} & \forall F_{k,l}, \forall P_q \rightsquigarrow P_r, & x_{q,r}^{k,l} \in \{0,1\}, \quad y_q^k \in \{0,1\} \\
\text{(1b)} & \forall T_k, & \sum_{P_q} y_q^k = 1 \\
\text{(1c)} & \forall F_{k,l}, \forall P_q \rightsquigarrow P_r, & x_{q,r}^{k,l} \le y_q^k \\
\text{(1d)} & \forall T_l, \forall F_{k,l}, \forall P_r, & y_r^k + \sum_{P_q \rightsquigarrow P_r} x_{q,r}^{k,l} \ge y_r^l \\
\text{(1e)} & \forall P_q, & \sum_{T_k} y_q^k w_{q,k} \le \tau \\
\text{(1f)} & \forall P_q \to P_r, & d_{q,r} = \sum_{\substack{P_s \rightsquigarrow P_t \text{ with} \\ P_q \to P_r \in P_s \rightsquigarrow P_t}} \left(\sum_{F_{k,l}} x_{s,t}^{k,l} \mathrm{data}_{k,l} \right) \\
\text{(1g)} & \forall P_q \to P_r, & \frac{d_{q,r}}{\mathrm{bw}_{q,r}} \le \tau \\
\text{(1h)} & \forall P_q & \sum_{P_q \to P_r \in E_P} \frac{d_{q,r}}{B_q^{\text{out}}} \le \tau \\
\text{(1i)} & \forall P_r & \sum_{P_q \to P_r \in E_P} \frac{d_{q,r}}{B_q^{\text{in}}} \le \tau
\end{array}
\right.
$$

Proof. We first prove that for any allocation of throughput ρ, described by x's and y's variables, $(x, y, \tau = 1/\rho)$ satisfies the constraints of the linear program.

- All tasks of the workflow are processed exactly once, thus Constraint (1b) is verified.
- In any allocation, $x_{q,r}^{k,l} = y_q^k \times y_r^l$, thus Constraint (1c) is verified.
- If $y_r^l = 1$, that is if P_r processes T_l, then P_r must own all files $F_{k,l}$. It can either have it because it also processed T_k (in this case $y_r^k = 1$) or because it received it from some processor P_q (and then $x_{q,r}^{k,l} = 1$). In both cases, Constraint (1d) is satisfied.
- As the allocation has throughput ρ, it means that the occupation time of each processor, each link, and each network interface is at most $1/\rho$, as explained in Section 4.3.

This is precisely what is stated by Constraints (1e), (1g), (1h) and (1i). Thus, these constraints are satisfied.

As all allocations satisfy the constraint of the linear program, and ρ_{opt} is the maximal value of the throughput under these constraints, then all allocations have a throughput smaller than, or equal to, ρ_{opt}.

We now prove that any solution of the linear program represents an allocation. We have to verify that all tasks are performed, all input data needed to process a task are correctly sent to the corresponding processor, and that the allocation has the expected throughput $1/\tau$.

- All tasks are processed exactly once due to constraint (1b).
- Thanks to Constraint (1c), a processor is allowed to send a file $F_{k,l}$ only if it processed T_k.
- Thanks to Constraint (1d), a task T_l with predecessor T_k is processed on P_r only if P_r either processed T_k, or received $F_{k,l}$ from some processor P_q.
- Thanks to Constraints (1e), (1g), (1h) and (1i), we know that the maximum utilization time of any resource (processor, link or network interface) is at least equal to τ, the corresponding allocation has a throughput at most $1/\tau$.

Thus, any solution of the linear program describes a valid allocation. In particular, there is an allocation with throughput ρ_{opt}. □

5.2 Single path, free routing

We now move to the free routing setting. The transfer of a given file between two processors can take any path between these processors in the platform graph. We introduce a new set of variables to take this into account. For any file $F_{k,l}$ and link $P_i \to P_j$, $f_{i,j}^{k,l}$ is an integer value, with value 0 or 1: $f_{i,j}^{k,l} = 1$ if and only if the transfer of file $F_{k,l}$ between the processor processing T_k to the one processing T_l takes the link $P_i \to P_j$. Using these new variables, we transform the previous linear program to take into account the free routing policy. The new program, Linear Program (2) has exactly the same equations than Linear Program (1) except for the following: 1) the new variables are introduced (Constraint (2a)); 2) the computation of the amount of data in Constraint (1f) is modified into Contraint (2f) to take into account the new definition of the routes; 3) the new set of constraints (2j) ensures that a flow of value 1 is defined by the variables $f^{k,l}$ from the processor executing T_k to the one executing T_k.

$$(2) \quad \left\{ \begin{array}{ll} \multicolumn{2}{l}{\text{MINIMIZE } \tau \text{ UNDER THE CONSTRAINTS}} \\ \text{(2a)} & \forall F_{k,l}, \forall P_q \rightsquigarrow P_r, x_{q,r}^{k,l} \in \{0,1\},\ y_q^k \in \{0,1\},\ f_{i,j}^{k,l} \in \{0,1\} \\ \text{(2f)} & \forall P_q \to P_r, d_{q,r} = \sum_{F_{k,l}} f_{i,j}^{k,l} \mathrm{data}_{k,l} \\ \text{(2j)} & \forall P_q, \forall F_{k,l}, \sum_{P_q \to P_r} f_{q,r}^{k,l} - \sum_{P_{r'} \to P_q} f_{r',q}^{k,l} = \sum_{P_t} x_{q,t}^{k,l} - \sum_{P_s} x_{s,q}^{k,l} \\ \multicolumn{2}{l}{\text{AND (1b), (1c), (1d), (1e), (1g), (1h), (1i)}} \end{array} \right.$$

In the following lemma, we clarify the role of the f variables.

Lemma 1. *Given a file $F_{k,l}$, the following two properties are equivalent*

$$\textit{(i)}\ \forall P_q, \quad \sum_{P_q \to P_r} f_{q,r}^{k,l} - \sum_{P_{r'} \to P_q} f_{r',q}^{k,l} = \begin{cases} 1 & \text{if } P_q = P_{\text{prod}} \\ -1 & \text{if } P_q = P_{\text{cons}} \\ 0 & \text{otherwise} \end{cases}$$

(ii) the set of links $P_q \to P_r$ such that $f_{q,r}^{k,l} = 1$ defines a route from P_{prod} to P_{cons}.

Proof. The result is straightforward since Property *(i)* is a simple conservation law of f quantities. Note that this route may include cycles. These cycles do not change the fact that the links can be used to ship the files from P_{prod} to P_{cons}, but the time needed for the transportation is artificially increased. That is why in our experiments these cycles are sought and deleted to keep routes as short as possible. □

The following theorem states that the linear program computes an allocation with optimal throughput: again, we denote $\rho_{\text{opt}} = 1/\tau_{\text{opt}}$, where τ_{opt} is the value of τ in any optimal solution of this linear program.

Theorem 3. *An optimal solution of Linear Program (2) describes an allocation with optimal throughput for the free routing policy.*

Proof. Similarly to the proof of Theorem 2, we first consider an allocation, define the x, y and f variables corresponding to this allocation, and prove that they satisfy the constraints of the linear program.

- Since $f_{i,j}^{k,l}$ describes if transfer $F_{k,l}$ uses link $P_i \to P_j$, all constraints excepted (2j) are satisfied by the same justifications as in Theorem 2.

- In any allocation, file $F_{k,l}$ must be routed from the processor executing T_k to the processor executing T_l (provided that these tasks are excepted by different processors).

Thus, f define a route between those two processors. Then, we note that

$$\sum_{P_t} x_{q,t}^{k,l} - \sum_{P_s} x_{s,q}^{k,l} = \begin{cases} 1 & \text{if } P_q \text{ executes } T_k \text{ and not } T_l \\ -1 & \text{if } P_q \text{ executes } T_l \text{ and not } T_k \\ 0 & \text{otherwise} \end{cases}$$

If T_k and T_k are executes on the same processor, all corresponding f and x variables are equal to 0. Thus, thanks to Lemma 1, all Constraints (2j) are verified.

We now prove that any solution of Linear Program (2) defines a valid allocation with throughput ρ under the Free routing policy.

- As in the proof of Theorem 2, we can prove that x and y variables define an allocation with throughput ρ.
- As above, we note that for a given file $F_{k,l}$

$$\sum_{P_t} x_{q,t}^{k,l} - \sum_{P_s} x_{s,q}^{k,l} = \begin{cases} 1 & \text{if } P_q \text{ executes } T_k \text{ and not } T_l \\ -1 & \text{if } P_q \text{ executes } T_l \text{ and not } T_k \\ 0 & \text{otherwise} \end{cases}$$

  If tasks T_k and T_l are not executed on the same processors, we know thanks to Lemma 1 that the f variables define a route from the processor executing T_k to the one executing T_l.

Thus, any solution of the linear program describes a valid allocation. In particular, there is an allocation with throughput ρ_{opt}. □

5.3 Multiple paths

Finally, we present our linear programming formulation for the most flexible case, the multiple-paths routing: any transfer may now be split into several routes in order to increase its throughput. The approach is extremely similar to the one used for the single route, free routing policy: we use the same set of f variables to define a flow from processors producing files to processors consuming them. The only difference is that we no longer restrict f to integer values: by using rational variables in $[0; 1]$, we allow each flow to use several concurrent routes. Theorem 4 expresses the optimality of the allocation found by the linear program. Its proof is very similar to the proof of Theorem 3.

$$(3) \quad \begin{cases} \text{MINIMIZE } \tau \text{ UNDER THE CONSTRAINTS} \\ \text{(3a)} \quad \forall F_{k,l}, \forall P_q \leadsto P_r, x_{q,r}^{k,l} \in \{0,1\},\ y_q^k \in \{0,1\},\ f_{i,j}^{k,l} \in [0;1] \\ \text{AND (1b), (1c), (1d), (1e), (2f), (1g), (1h), (1i), (2j)} \end{cases}$$

The role of the f variable is modified, as expressed by the following lemma.

Lemma 2. *Given a file $F_{k,l}$, the following two properties are equivalent*

$$\textit{(i)}\ \forall P_q \quad \sum_{P_q \to P_r} f^{k,l}_{q,r} - \sum_{P_{r'} \to P_q} f^{k,l}_{r',q} = \begin{cases} 1 & \text{if } P_q = P_{\text{prod}} \\ -1 & \text{if } P_q = P_{\text{cons}} \\ 0 & \text{otherwise} \end{cases}$$

(ii) the set of links $P_q \to P_r$ such that $f^{k,l}_{q,r} \neq 0$ defines a set of weighted routes from P_{prod} to P_{cons}, with total weight 1.

Proof. Consider the subgraph of the platform graph comprising only the links$P_q \to P_r$ such that $f^{k,l}_{q,r} \neq 0$. We construct the set of weighted routes by the following iterative process. We extract a route r from P_{prod} to P_{cons} from this graph (such a route exists thanks to the conservation law). We then compute the minimum weight w of the links in route r. Route r is added with weight w to the set of weighted routes, and the subgraph is pruned as followed: w is subtracted from the value of $f^{k,l}$ of all links included in route r, and links whose $f^{k,l}$ value becomes null are removed from the subgraph. We can prove that Property *(i)* still holds with value $1 - w$ instead of 1. We continue the process until there is no more link in the subgraph. □

Theorem 4. *An optimal solution of Linear Program (3) describes an allocation with optimal throughput for the multiple paths policy.*

6 Performance evaluation

In this section, we present the simulations performed to study the performance of our strategies. Simulations allow us to test different heuristics on the very same scenarios, and also to consider far more scenarios than real experiments would. We can even test scenarios that would be quite hard to run real experiments with. This is especially true for the flexible or multiple-path routing policies. Our simulations consist here in computing the throughput obtained by a given heuristic on a given platform, for some application graphs. We also study another metric: the latency of the heuristics, that is the time between the beginning and the end of the processing of one input data set. A large latency may lead to a bad quality of service in the case of an interactive workflow (e.g., in image processing), and to a huge amount of temporary files. This is why we intend to keep the latency low for all input data sets.

6.1 Reference heuristics

In order to assess the quality and usefulness of our strategies, we compare them against four classical task-graph scheduling heuristics. Some of these heuristics (Greedy and HEFT) are dynamic strategies: they allocate resources to tasks in the order of their arrival. As this approach is not very practical when scheduling a series of identical workflows, we transform these heuristics into static scheduling strategies: the mapping of a single allocation is computed with the corresponding strategy (Greedy or HEFT), and then this allocation is used in a pipelined way for all instances.

Greedy. This strategy maps the tasks onto the platform starting from the task with the highest $w_{i,k}$ value, i.e., the task which can reach the worst computation time. The processor that would process this task the fastest is then allocated to this task, and the processor is "reserved" for the time needed for the processing. The allocation proceeds until all tasks are scheduled. Communications are scheduled using either the compulsory route, or the shortest path between allocated processors in case of flexible routing.

HEFT. This heuristic builds up an allocation for a single instance using the classical Heterogeneous Earliest Finish Time [35]. Then, this allocation is used for all instances.

Pure data-parallelism. We also compare our approach to a pure data-parallelism strategy: in this case, all tasks of a given instance are processed sequentially on a given processor, as detailed in the introduction.

Multi-allocations upper bound. In addition to the previous classical heuristics, we also study the performance when mixing control- and data-parallelism. This approach uses concurrent allocations to reach the optimal throughput of the platform, as is explained in details in [6]. Rather than using the complex algorithm described in that paper for task graphs with bounded dependences, we use an upper bound on the throughput based on this study, which consists of a simple linear program close to the one described in this paper, and solved over the rational numbers. This bound is tight when the task graph is an in- or out-tree, but may not take all dependences into account otherwise. This upper bound, however, has proved to be a very good comparison basis in the following, and is used as a reference to assess the quality of other heuristics. No latency can be derived from this bound on the throughput, since no real schedule is constructed.

6.2 Simulation settings

Platforms. We use several platforms representing existing computing Grids. The descriptions of the platforms were obtained through the SimGrid simulator repository [1], as described in [20].

- DAS-3, the Dutch Grid infrastructure [14],
- Egee, a large-scale European multi-disciplinary Grid infrastructure, gathering more than 68.000 CPUs [16],
- Grid5000, a French research Grid which targets 5000, processors [9],
- GridPP, the UK Grid infrastructure for particle physics [10].

Users are often limited to a small number of processors of a given platform. To simulate this behavior, a subset of the available processors is first selected and then used by all heuristics. It is composed of 10 to 20 processors. To evaluate our fixed-routing strategies, we pre-compute a shortest-path route between any pair of processors, which is used as the compulsory route.

Applications. Several workflows are used to assess the quality of our strategies, with a number of tasks between 8 and 12: 1) pipeAlign [31], a protein family analysis tool, 2) several random task graphs generated by the TGFF generator [15]. In order to evaluate the impact of communications on the quality of the result, we artificially multiply by different factors all communications of the application graphs. There are many ways to define a communication-to-computation ratio (CCR) for a given workflow. We chose to define an average computation time t_{comp} by dividing the sum of all tasks by the average computational power of the platform (excluding powerless nodes like routers), and an average communication time t_{com} by dividing the sum of all files by the average bandwidth. Then the CCR is given by the ratio $t_{\text{com}}/t_{\text{comp}}$. Finally, we impose the first task and the last one to be processed on the first processor. These tasks have a size 0 and correspond to the storage of input and output data.

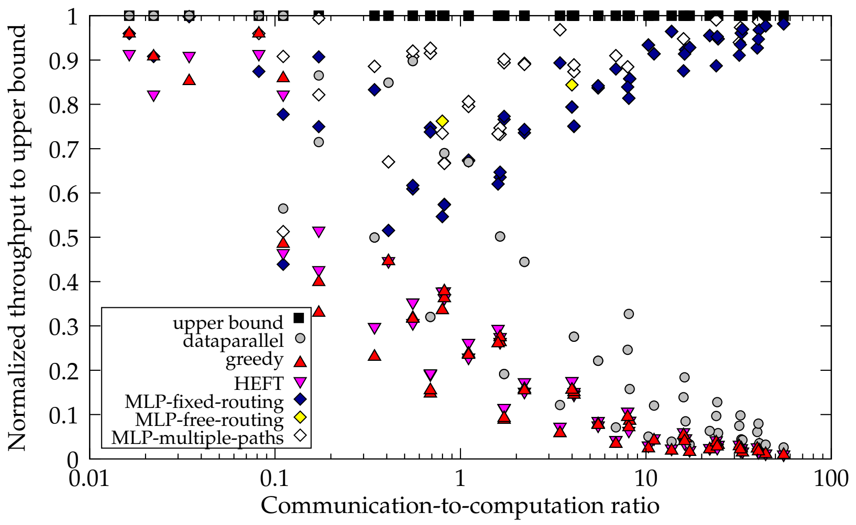
6.3 Results

6.3.1 Throughput and Latency

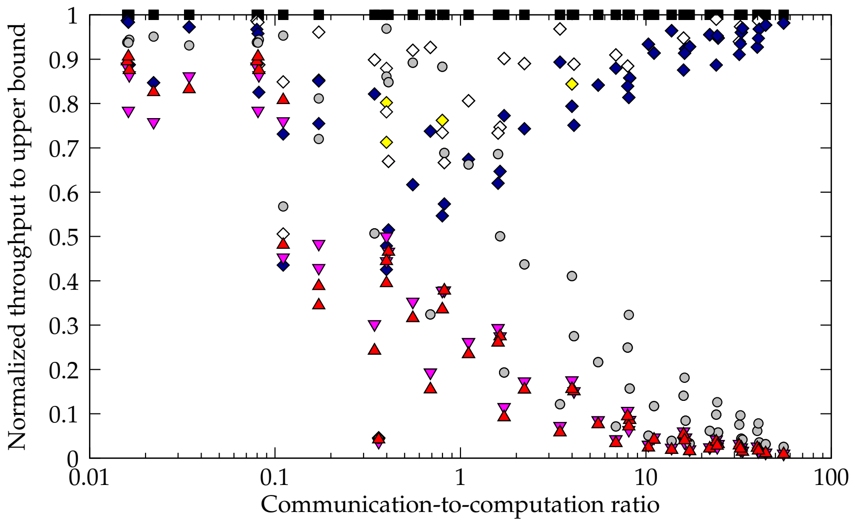
As said before, we compare both throughput and latency for each solution. For each of the 233 application/platform scenarios, results are normalized to the best one: the performance of a heuristic on an instance is divided by the best performance achieved by a heuristic on that instance (or by the upper bound for throughput); therefore, the closer to 1, the better. Figure 4(a) gives the throughput observed for related applications, while Figure 4(b) gives the throughput for unrelated applications. Latencies are displayed for related applications in Figure 5(a) and for unrelated applications in Figure 5(b). The two last figures Figure 6(a) and Figure 6(b) show the latency divided by τ, which is the inverse of the throughput and represents the time between the completion of two successive instances. This gives an idea on the performance for the latency: the smaller the ratio, the shorter the latency in comparison to τ. Normalizing with τ allows us to compare latencies of very different scenarios, in spite of the discrepancies in computing sizes. Note that this ratio also gives the average number of instances that are being processed at any time (started but not completed), and is thus an indication of the storage capacity needed for temporary files.

In the following, our strategies based on Mixed Linear Programing are noted MLP in the following.

Data-parallel. When dealing with the related model and a low CCR, the data-parallel strategy offers the best throughput since all processors are working at full speed. Since

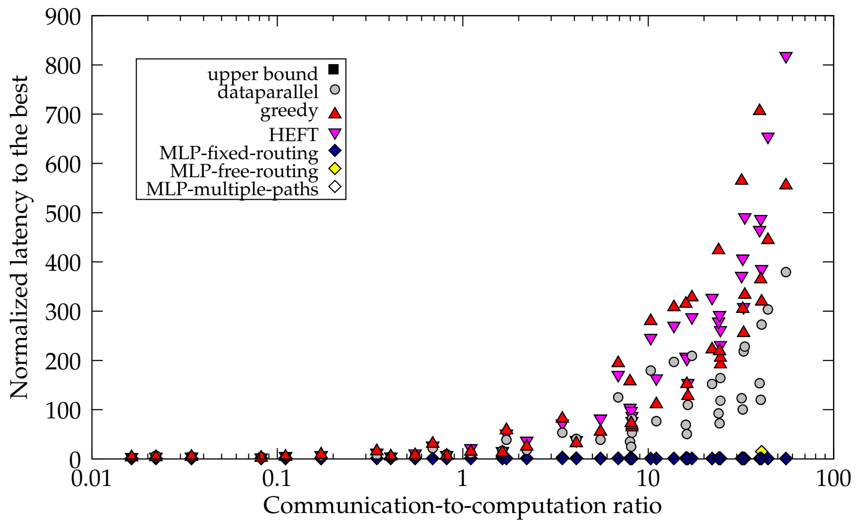


(a) Throughputs for related applications

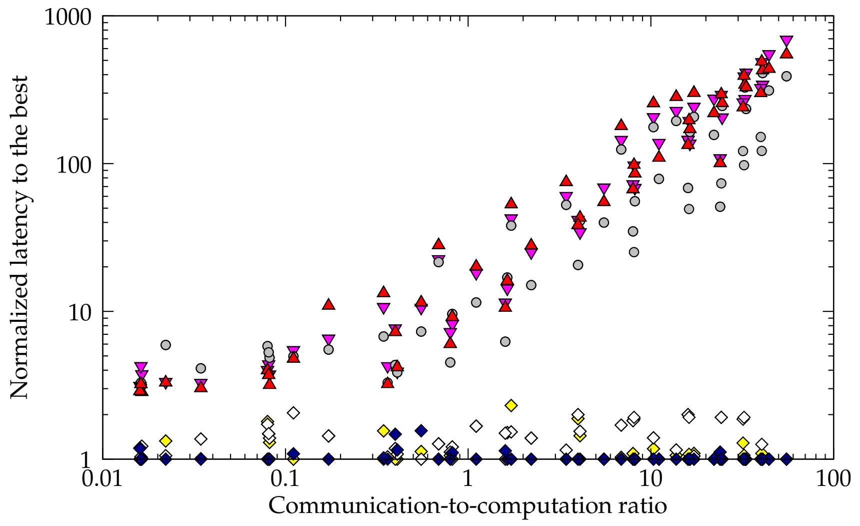


(b) Throughputs for unrelated applications

Figure 4: Normalized throughputs of different strategies, with related and unrelated applications.

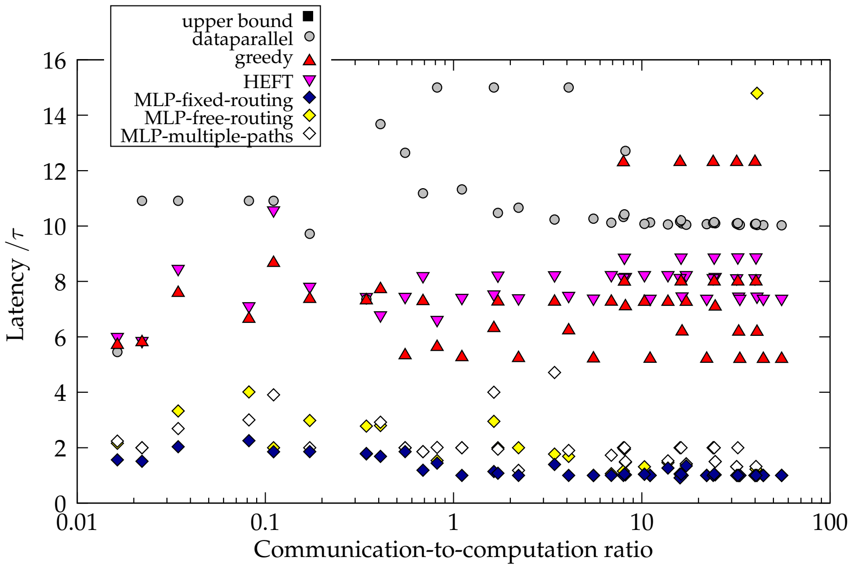


(a) Latencies for related applications

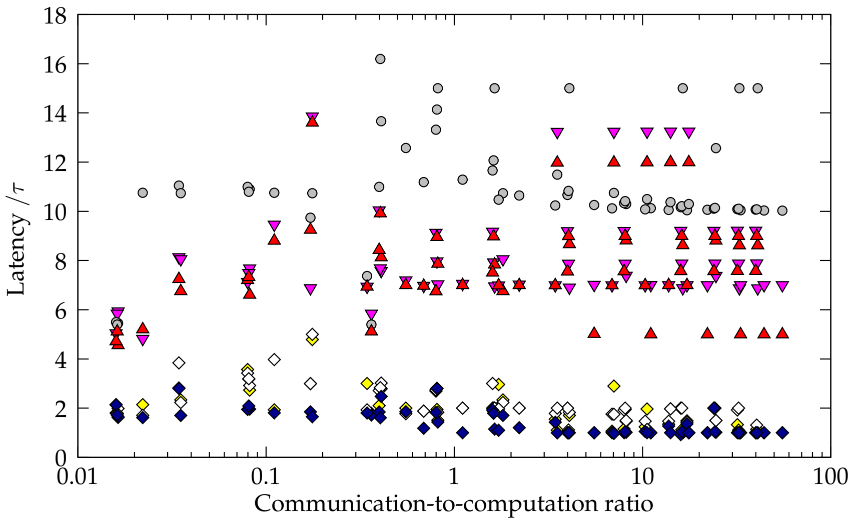


(b) Latencies for unrelated applications

Figure 5: Normalized latencies of different strategies, with related and unrelated applications.



(a) Ratio between latency and τ for related applications



(b) Ratio between latency and τ for unrelated applications

Figure 6: Ratio between latency and τ for different strategies, with related and unrelated applications.

sending input and output data requires communications, performance decreases with the CCR, down to 10% of the upper bound for a CCR greater than 10. When processing DAGs with unrelated computing speeds, this strategy can be the worst even with low CCRs, because other strategies take unrelated speeds into account. Moreover, as it makes use of all processors, even slow ones, the data-parallel heuristic leads to a very high latency, which increases with the CCR. In Figure 6(a) and Figure 6(b), we can see that the ratio between latency and τ (which is also the average number of unfinished instances in the system) is roughly equal to the number of used processors, which is a natural consequence of the data-parallel strategy.

Greedy and HEFT. In both related and unrelated cases, HEFT and greedy heuristics have rather good results when the CCR is very low: around 80% of the upper bound for ratios below 0.1. When the scenario becomes more communication-intensive (CCR equal to, or greater than, 1), their performance is dropping to 30% of the upper bound, and even to 10% when CCR ≥ 10. On the other hand, these strategies build schedules whose latency is often very large compared to those of own MLP strategies. Like the data-parallel strategy, relative latencies increase with the CCR, leading to latencies 100 times worse the best one when the CCR is greater than 10. The ratio between latency and τ is better than for the data-parallel strategy but roughly 3 times higher than our MLP strategies.

MLP strategies. Our three strategies based on linear programs often return similar results: the best one is MLP-multiple-paths, followed by MLP-free-routing, and MLP-fixed-routing. This is quite natural: the more flexible the routing, the better the results. For CCRs below 10, these strategies obtain throughputs between 50% and 80% of the upper bound, and their performance increases with the CCR: when this ratio is over 10, our strategies achieve more than 80% of the upper bound, and often over 90%. Moreover, the MLP strategies always give the best latency. MLP-fixed-routing almost always achieves the best latency: using other routes than the shortest-path, or concurrent routes, can increase the latency (up to a factor 2). Finally, the ratio between latency and τ is very small, often between 2 and 4, less than other strategies.

6.3.2 Running time of the algorithm

Our strategies relies on mixed linear programs, which can take much time to solve. Even if the use of linear programs in our heuristics can significantly slow down the computation of the schedule, it allows to reach better throughputs, and thus better running times. When dealing with very large series of workflows, this gain can be significant and justify the choice of a scheduler. Moreover, specialized tools like GPLK [24] or CPLEX [13] can solve mixed linear programs efficiently. During our simulations, all linear programs and mixed linear programs were solved using CPLEX 11.0 on a 2.4GHz Opteron processor. As we target reasonable numbers of tasks, the processing time of the schedule is kept low: all problems were solved in less than 10 seconds.

7 Conclusion and perspectives

In this paper, we have studied the scheduling of a series of workflows on a heterogeneous platform. We have taken advantage of the regularity due to the series to optimize the system throughput by applying steady-state techniques. We have derived single-allocation strategies, which combine good performance with simple control. Indeed, we have proven that our mixed linear program computes an optimal allocation under a number of routing scenarios. Simulations have proven that the benefit of our approach in comparison to classical reference heuristics is significant as soon as communication times are not negligible, and that our allocations lead to much smaller latencies as a side effect. Future work include simplifying our mixed linear program to cope with larger applications, and using task duplication to further improve the system throughput.

References

[1] SimGrid: a Generic Framework for Large-Scale Distributed Experimentations. Henri casanova and arnaud legrand and martin quinson. In *Proceedings of the 10th IEEE International Conference on Computer Modelling and Simulation (UKSIM/EUROSIM'08)*, 2008.

[2] Ishfaq Ahmad and Yu-Kwong Kwok. On exploiting task duplication in parallel program scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 9(9):872–892, 1998.

[3] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Scheduling strategies for master-slave tasking on heterogeneous processor platforms. *IEEE Trans. Parallel Distributed Systems*, 15(4):319–330, 2004.

[4] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Bandwidth-centric allocation of independent tasks on heterogeneous platforms. In *International Parallel and Distributed Processing Symposium (IPDPS'2002)*. IEEE Computer Society Press, 2002.

[5] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Scheduling strategies for mixed data and task parallelism on heterogeneous clusters. *Parallel Processing Letters*, 13(2), 2003.

[6] 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. Technical report, LIP, ENS Lyon, France, apr 2004. Also available as INRIA Research Report RR-5198.

[7] Olivier Beaumont, Arnaud Legrand, Loris Marchal, and Yves Robert. Steady-state scheduling on heterogeneous clusters. *Int. J. of Foundations of Computer Science*, 16(2):163–194, 2005.

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

[9] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frederic Desprez, Emmanuel Jeannot, Yvon Jégou, Stéphane Lanteri, Julien Leduc, Noredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi, and Touché Irena. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, November 2006.

[10] David Britton, Tony Cass, Peter Clarke, Jeremy Coles, Tony Doyle, NeilGeddes, and John Gordon. GridPP: Meeting the particlephysics computing challenge. In *UK e-Science All Hands Conference*, 2005.

[11] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven 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.

[12] P. Chrétienne, E. G. Coffman Jr., J. K. Lenstra, and Z. Liu, editors. *Scheduling Theory and its Applications*. John Wiley and Sons, 1995.

[13] Ilog cplex: High-performance software for mathematical programming and optimization. `http://www.ilog.com/products/cplex/`.

[14] The distributed asci supercomputer 3 (DAS-3). `http://www.cs.vu.nl/das3/`.

[15] R.P. Dick, D.L. Rhodes, and W. Wolf. Tgff task graphs for free. *codes*, 00:97, 1998.

[16] Enabling grids for e-science (EGEE). `http://www.eu-egee.com/`.

[17] Lionel Eyraud-Dubois, Arnaud Legrand, Martin Quinson, and Frédéric Vivien. A first step towards automatically building network representations. In *Proceedings of Euro-Par 2007*, volume 4641 of *LNCS*, pages 160–169, 2007.

[18] I. Foster. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2004.

[19] Ian T. Foster, Markus Fidler, Alain Roy, Volker Sander, and Linda Winkler. End-to-end quality of service for high-end applications. *Computer Communications*, 27(14):1375–1388, 2004.

[20] Marc-Eduard Frincu, Martin Quinson, and Frédéric Suter. Handling very large platforms with the new simgrid platform description formalism. Technical Report 0348, INRIA, feb 2008.

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

[22] Cécile Germain, Vincent Breton, Patrick Clarysse, Yann Gaudeau, Tristan Glatard, Emmanuel Jeannot, Yannick Legré, Charles Loomis, Isabelle Magnin, Johan Montagnat, Jean-Marie Moureaux, Angel Osorio, Xavier Pennec, and Romain Texier. Grid-enabling medical image analysis. *Journal of Clinical Monitoring and Computing*, 19(4–5):339–349, October 2005.

[23] Tristan Glatard, Johan Montagnat, Diane Lingrand, and Xavier Pennec. Flexible and efficient workflow deployement of data-intensive applications on grids with MOTEUR. *International Journal of High Performance Computing and Applications*, 2008.

[24] Glpk (gnu linear programming kit). `http://www.gnu.org/software/glpk/`.

[25] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *HiPC*, pages 197–208, 2007.

[26] Marc Daniel Haunschild, Bernd Freisleben, Ralf Takors, and Wolfgang Wiechert. Investigating the dynamic behavior of biochemical networks using model families. *Bioinformatics*, 21(8):1617–1625, 2005.

[27] B. Hong and V.K. Prasanna. Distributed adaptive task allocation in heterogeneous computing environments to maximize throughput. In *International Parallel and Distributed Processing Symposium IPDPS'2004*. IEEE Computer Society Press, 2004.

[28] Soojin Lee, Min-Kyu Cho, Jin-Won Jung, and Jai-Hoon Kim nd Weontae Lee. Exploring protein fold space by secondary structure prediction using data distribution method on grid platform. *Bioinformatics*, 20(18):3500–3507, 2004.

[29] Thomas M. Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, R. Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

[30] Michael A. Palis, Jing-Chiou Liou, and David S. L. Wei. Task clustering and scheduling for distributed memory parallel architectures. *IEEE Trans. Parallel Distrib. Syst.*, 7(1):46–55, 1996.

[31] Pipealign. `http://bips.u-strasbg.fr/PipeAlign/Documentation/`.

[32] Joe Pitt-Francis, Alan Garny, and David 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.

[33] B. A. Shirazi, A. R. Hurson, and K. M. Kavi. *Scheduling and load balancing in parallel and distributed systems*. IEEE Computer Science Press, 1995.

[34] Frédéric Suter, Frederic Desprez, and Henri Casanova. From heterogeneous task scheduling to heterogeneous mixed parallel scheduling. In *Euro-Par*, pages 230–237, 2004.

[35] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Task scheduling algorithms for heterogeneous processors. In *HCW '99: Proceedings of the Eighth Heterogeneous Computing Workshop*, page 3, Washington, DC, USA, 1999. IEEE Computer Society.

[36] Vasily Volkov and James Demmel. Lu, qr and cholesky factorizations using vector capabilities of gpus. Technical Report UCB/EECS-2008-49, EECS Department, University of California, Berkeley, May 2008.

[37] R. Wolski, N.T. Spring, and J. Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(10):757–768, 1999.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399