2008 14th IEEE International Conference on Parallel and Distributed Systems

Allocating Series of Workflows on Computing Grids

Matthieu Gallet [2,4,5] Loris Marchal [1,4,5] Frédéric Vivien [3,4,5]
[1] CNRS [2] ENS Lyon [3] INRIA [4] Université de Lyon
[5] LIP laboratory, UMR 5668, ENS Lyon – CNRS – INRIA – UCBL, Lyon, France
{matthieu.gallet,loris.marchal,frederic.vivien}@ens-lyon.fr

Abstract

In this paper, 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. The 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.*

Keywords: Workflows, DAGs, scheduling, heterogeneity, computing Grid.

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 [10], etc. Distributing an application on such a platform is a complex duty. As far as performance is concerned, we have to take into account the computing requirements of each task, the communication volume of each data transfer, as well as the platform heterogeneity: the processing resources are intrinsically heterogeneous, and run different systems and middlewares; the communication links are heterogeneous as well, due to their various bandwidths and congestion status. 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, independent parameter sets, or independent models. We thus 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 it (Section 3) and describe our solutions (Section 4). Finally, we assess the quality of these solutions through simulations (Section 5) and conclude (Section 6). Due to lack of space, we refer the interested reader to the companion research report [9] for a comprehensive review of related work.

2. Problem motivation

2.1. Dynamic 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,

1521-9097/08 $25.00 © 2008 IEEE
DOI 10.1109/ICPADS.2008.96

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 no 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.

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 [15] 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 data parallelism. Then, 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 [1, 2] 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. 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 a *steady-state* approach: 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. Notations, hypotheses, and complexity

We now detail our model and assess the problem complexity.

3.1. Platform and application model

We denote by $G_P = (V_P, E_P)$ the undirected graph representing the platform, where $V_P = \{P_1, \ldots, P_p\}$ 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 $P_q \rightarrow P_r$ 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. 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.

We use a bidirectional 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.

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 $P_q \rightarrow P_r$ takes a time $\frac{\mathrm{data}_{i,j}}{\mathrm{bw}_{q,r}}$. Figure 1(b) gives an example of application graph. Computation task T_k needs a time $w_{i,k}$ to be entirely processed by processor P_i. This last notation corresponds to the so-called unrelated-machine model: a processor can be fast for a given type of tasks 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.

3.2. 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 associating:*

- *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 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 [8].

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.

The three routing policies allow us to model a wide range of practical situations, current and future. The techniques exposed in Section 4 enable us to deal with any of these models 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.

3.3. 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 $\rho = \lim_{t \rightarrow \infty} \frac{N(t)}{t}$.*

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

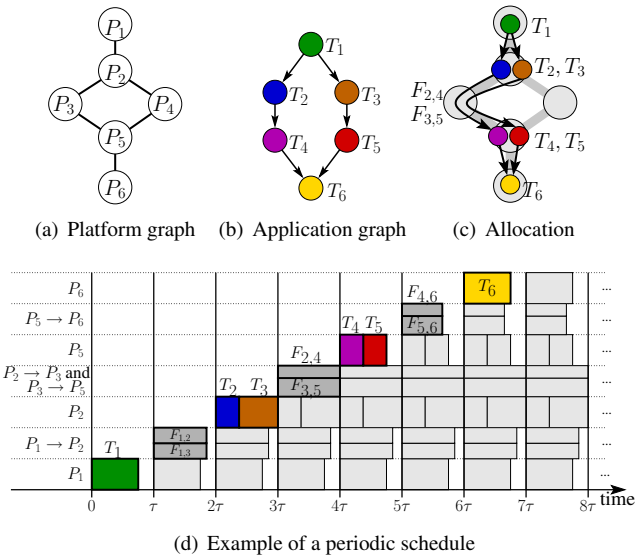
Upper bound on the achievable throughput. First, we consider the time spent by each resource on one instance of a given allocation σ. In other words, we consider the time spent by each resource for processing a single copy of our workflow under allocation σ.

- 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 $d_{q,r} = \sum_{(i,j),P_q \to P_r \in \sigma(F_{i,j})} \text{data}_{i,j}$ for single-path policies, and $d_{q,r} = \sum_{F_{i,j}} \sum_{\substack{(w_a,P_a)\in\sigma(F_{i,j}) \\ P_q \to P_r \in P_a}} w_a \times \text{data}_{i,j}$ for 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 P_q outgoing interface: $t_q^{\text{out}} = \sum_r d_{q,r}/B_q^{\text{out}}$;
  - on P_q incoming interface: $t_q^{\text{in}} = \sum_r d_{r,q}/B_q^{\text{in}}$.

We can now compute the maximum time τ spent by any resource for the processing of one instance: $\tau = \max\left\{\max_{P_q}\{t_q^{\text{comp}}, t_q^{\text{out}}, t_q^{\text{in}}\}, \max_{P_q \to P_r} t_{q,r}\right\}$. This gives us an upper bound on the achievable throughput: $\rho \leq \rho_{\max} = 1/\tau$. 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.

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 [1], and a fully comprehensive proof can be found in [2].

Figure 1 illustrates how to build a periodic schedule of period τ for the workflow described on Figure 1(b), on the platform of Figure 1(a), using the allocation of Figure 1(c).



(a) Platform graph (b) Application graph (c) Allocation

(d) Example of a periodic schedule

Figure 1. Example of periodic schedule. Only the first instance is represented with task and file labels.

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 $1/\tau$.

3.4. NP-completeness of throughput optimization

We now formally define the decision problem associated to the problem of maximizing the throughput. The proof of this result is a simple reduction from Minimum Multiprocessor Scheduling, and is available in the companion research report [9].

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.*

4. 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.

4.1. 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 5 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.

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

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 3.3.

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. Due to lack of space, the proof of this result is available in the companion research report [9].

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

4.2. 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 constraints than Linear Program (1) except for the following: (i) the new variables are introduced (Constraint (2a)); (ii) the computation of the amount of data in Constraint (1f) is modified into Constraint (2f) to take into account the new definition of the routes; (iii) 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.

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

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. As previously, all the proofs are available in the companion research report [9]. The proof is based on the following remark: for a given file $F_{k,l}$, proving that the links $P_q \to P_r$ such that $f_{q,r}^{k,l} = 1$ define a route from some processor P_{prod} to some other processor P_{cons} is equivalent to proving that for all processor P_q, $\sum_{P_q \to P_r} f_{q,r}^{k,l} - \sum_{P_{r'} \to P_q} f_{r',q}^{k,l}$ is equal to 1 if $Ppq = P_{\text{prod}}$, -1 if $P_q = P_{\text{cons}}$ and 0 otherwise.

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

4.3. 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.

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

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

5. 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.

5.1. 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 [14]. 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 [2]. 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.

5.2. Simulation settings

Platforms. We use several platforms representing existing computing Grids. The descriptions of the platforms were obtained through the SimGrid simulator repository [3]:

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

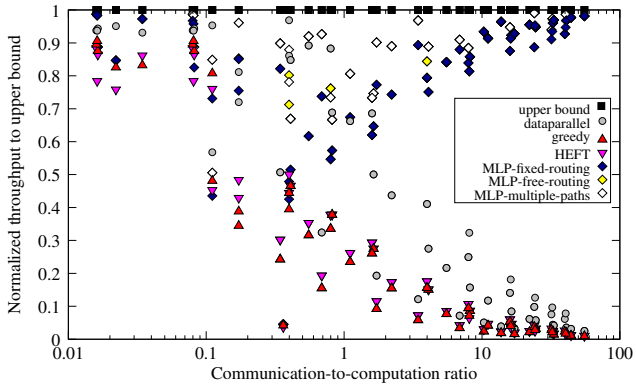
Users are often limited to a small number of processors of a given platform. To simulate this behavior, a subset of the available processors is first selected and then used by all heuristics. It is composed of 10 to 20 processors. To evaluate our fixed-routing strategies, we pre-compute a shortest-path 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: (i) pipeAlign [13], a protein family analysis tool, (ii) several random task graphs generated by the TGFF generator [6]. 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 choose 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. Our application setting includes both related and unrelated application, as discussed in Section 3.1, but we do not distinguish them as they lead to comparable results.

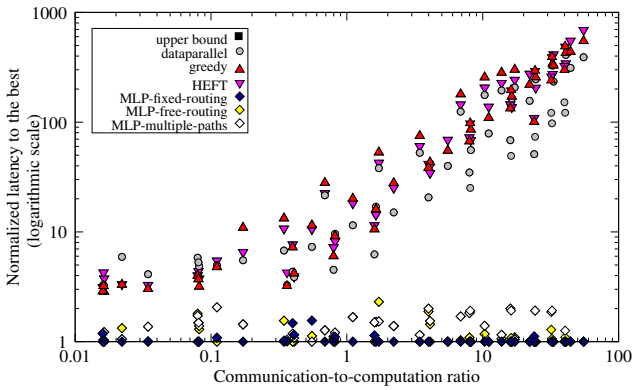
5.3. Results

5.3.1 Throughput and Latency

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 2(a) gives the observed throughput, while Figure 2(b) presents the latencies. The strategies based on Mixed Linear Programing are noted MLP in the following.



(a) Throughput of the applications



(b) Latencies of the applications

Figure 2. Performance of different strategies.

Data-parallel. When dealing with a low CCR, the data-parallel strategy offers the best throughput since all processors are working at full speed. Since 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. Moreover, as it makes use of all proces-

sors, even slow ones, the data-parallel heuristic leads to a very high latency, which increases with the CCR.

Greedy and HEFT. The reference 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.

MLP strategies. The 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%. MLP-fixed-routing almost always achieves the best latency: using other routes than the shortest-path, or concurrent routes, can slightly increase the latency (up to a factor 2).

5.3.2 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 [11] or CPLEX [5] 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.

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

[2] 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. Research report, LIP, ENS Lyon, France, Apr. 2004.

[3] H. Casanova, A. Legrand, and M. Quinson. SimGrid: a generic framework for large-scale distributed experimentations. In *Proceedings of IEEE UKSIM/EUROSIM'08*, 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] R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: task graphs for free. In *CODES*, pages 97–101, 1998.

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

[8] I. T. Foster, M. Fidler, A. Roy, V. Sander, and L. Winkler. End-to-end quality of service for high-end applications. *Computer Communications*, 27(14):1375–1388, 2004.

[9] M. Gallet, L. Marchal, and F. Vivien. Allocating series of workflows on computing grids. Research report RR-6603, INRIA, 2008.

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

[11] GLPK (GNU Linear Programming Kit). `http://www.gnu.org/software/glpk/`.

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

[14] 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.

[15] R. Wolski, N. 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.