Efficient Scheduling of Task Graph Collections on Heterogeneous Resources

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* collection of task graphs *problem. We are looking for a competitive scheduling algorithm not requiring complex control. We thus only consider single-allocation strategies. In addition to a mixed linear programming approach to find an optimal allocation, we present different heuristic schemes. Then, using simulations, we compare the performance of our different heuristics to the performance of a classical scheduling policy in Grids,* HEFT. *The results show that some of our static-scheduling policies take advantage of their platform and application knowledge and outperform* HEFT*, especially under communication-intensive scenarios. In particular, one of our heuristics,* DELEGATE*, almost always achieves the best performance while having lower running times than* HEFT.

Keywords Workflows, DAGs, scheduling, steady state, heterogeneity, computing Grids.

1 Introduction

Computing Grids gather large-scale distributed and heterogeneous resources, and make them available to large communities of users [13]. 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 [8], bioinformatics [21], medical image processing [17], 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 sometimes 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* made of a large collection of input data sets that must all be processed by the same application. We thus 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 [20] or independent parameter sets [24]. We thus concentrate on how to map several instances of a same task graph onto a computing platform, that is, on how to decide on which resource each instance of each task has to be processed.

We start by motivating the problem and describing related work (Section 2), then we formally define the problem (Section 3) and describe our solutions (Section 4). Finally, we assess the quality of these solutions through simulations (Section 5) and conclude (Section 6).

2 Problem motivation and related work

2.1 Workflow scheduling in Grids

In Grid computing, jobs are often organized as sets of tasks with dependences, thus resulting in a *task graph*, sometimes also called a *workflow*. Scheduling task graphs on Grids is the subject of a wide literature, and many

978-1-4244-3750-4/09/$25.00 ©2009 IEEE

tools exist to manage and schedule such workflows, such as MOTEUR [18]. These tools usually include scheduling heuristics to map the workflow tasks onto the available resources. These heuristics were often inherited from the Direct-acyclic graph (DAG) scheduling literature and were more or less adapted to cope with the intrinsic heterogeneity of Grid environments. The most used techniques are list scheduling [9], clustering [22], and task duplication [1]. The most famous of all task graph scheduling algorithms for Grids is certainly HEFT [26]. We will use it as our baseline reference.

2.2 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, 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 the same task graph. First, we can take advantage of the regularity of our collection of task graphs. Second, we assume the computing platform to be stable, and processors to be dedicated to our application. We can then use performance measurement tools like NWS [28] in order to get information on machine speeds and link bandwidths. Also, there is no external workload we should take into account. Such a situation arises, for instance, when one is trying to make the best of one's reservation on a Grid.

We claim that by taking advantage of such knowledge, we can achieve better results using *static* scheduling techniques, that is, by anticipating the mapping and scheduling of the whole workload at its submission date. The simulation experiments (Section 5) will show that we achieve significantly better performance than a classic scheduler like HEFT [26].

2.3 Steady-state scheduling

Our objective is to take advantage of the regularity of the problem, and to make use of *steady-state* scheduling techniques. In this paper we consider a Grid job made of a large number of data sets which have to be processed using the same task graph. We can 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. For problems composed of a large number of data sets, optimizing the steady-state phase allows to derive efficient, near-optimal schedules. In scheduling, the classical objective is to minimize the running time of the job, or *makespan*, which is an NP-hard problem in most practical situations [16, 25, 9]. However, by using steady-state techniques, we relax this objective and concentrate on maximizing the system throughput. Fortunately, it turns out that the optimal steady-state schedule can often be characterized very efficiently.

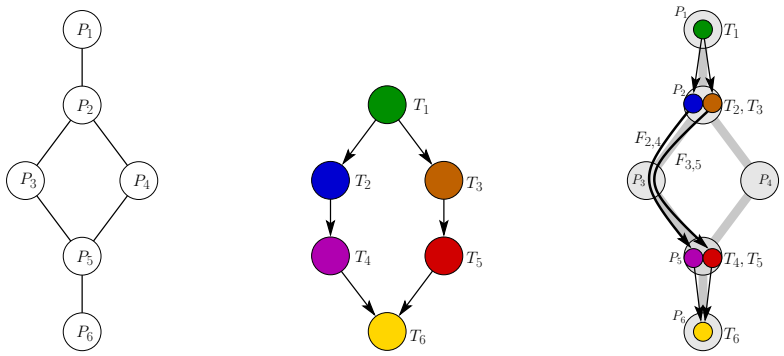
The steady-state approach has been pioneered by Bertsimas and Gamarnik [6]. This approach has been successfully applied to problems as diverse as network packet routing [6], pipelining broadcasts [4], or scheduling independent tasks [2] or divisible loads [19] on heterogeneous platforms. A steady-state approach for scheduling collections of identical task graphs was proposed in [5]. The solution of [5], although asymptotically optimal under reasonable assumptions, may not be practical. Indeed, this solution may use a large number of different allocations and thus require a lot of control. Furthermore, the schedules built by [5] may have large initialization phases and periods. They may thus require a large number of task graph instances before the steady-state can be reached. This is why we are targeting steady-state schedules using a single allocation. In [14] we have addressed the theoretical part of this problem: establishing its complexity and proposing mixed-linear programs to solve it (we recall these results in Sections 3.4 and 4.1). In the current paper we aim at designing practical heuristics.

3 Notations, hypotheses, and 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 $P_q \to P_r$ 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. 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, so long as the bandwidth limitation is not exceeded on links, either on incoming or on outgoing ports.



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

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 $P_q \rightarrow P_r$ 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. 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 [27]. Unrelated performance may also come from memory requirements. Indeed, a given task requiring a lot of memory may 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.

3.2 Allocations

As described in the introduction, we assume that we have a large collection of input data sets to process. 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 task graph. For the sake of simplicity, we are looking for strategies where all tasks of a given type T_i are processed on the same resource, which means that the allocation of tasks to processors is the same for all instances. The goal of this restriction is to produce practical schedules requiring far less control than classical steady-state solutions (see the related work discussion). We now formally define what an allocation is.

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 $\sigma(T_j)$. The path for any transfer from P_q to P_r is fixed a priori. 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.*

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.

Definition of throughput We focus on the optimization of the steady state. Thus, we are not interested in minimizing the overall execution time for a given number of task graph 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. We are only interested in very specific schedules, consisting of a single 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.

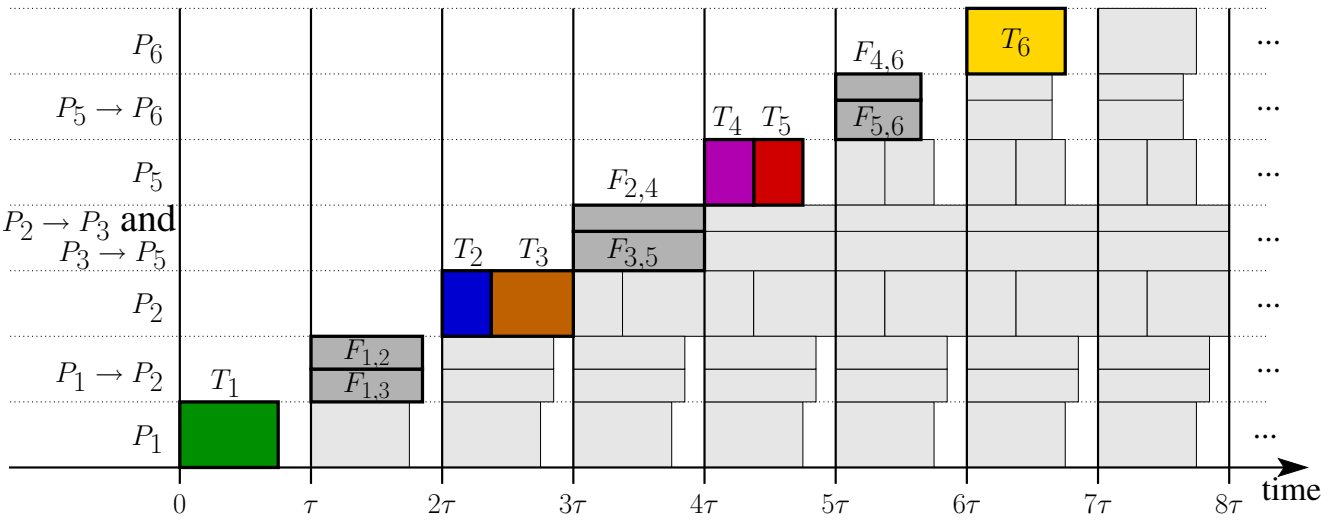


Figure 2. A periodic schedule for the allocation of Figure 1(c). Only the first instance is represented with task and file labels.

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 task graph under allocation σ.

- The computation time spent by 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 $d_{q,r} = \sum_{(i,j),P_q \to P_r \in \sigma(F_{i,j})} \text{data}_{i,j}$. 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's outgoing interface: $t_q^{\text{out}} = \sum_r d_{q,r}/B_q^{\text{out}}$;
  - on P_q's 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 [3], and a fully comprehensive proof can be found in [5].

Figure 2 illustrates how to build a periodic schedule of period τ for the task graph described on Figure 1(b), on the platform of Figure 1(a), using the allocation of Figure 1(c). 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 [15].

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.*

4 Finding an allocation with good throughput

4.1 Optimal allocation trough mixed linear programming

Here, we recall how to compute the optimal allocation [14]. In that paper, we proposed a method, using mixed linear programming: we express the problem as a linear program where integer and rational variables coexist. Although some softwares can solve such linear programs [11], the problem remains NP-complete and this method can only be used for small task graphs (up to 12 tasks with our current implementation). However, this method provides an interesting upper bound on the achievable throughput to compare other solutions to on small task graphs.

Our linear programming formulation makes use of both integer and rational variables. The integer variables are described below. They can only take values 0 or 1.

- 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 express the problem as a set of linear constraints.

The objective function is to minimize the maximum time τ spent on any resource, 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 that task.
- 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.

$$
\left\{
\begin{array}{ll}
\multicolumn{2}{l}{\text{MINIMIZE } \tau \text{ UNDER THE CONSTRAINTS}} \\
\text{(1a)} & \forall F_{k,l}, \forall P_q \rightsquigarrow P_r, x_{q,r}^{k,l} \in \{0,1\}, y_q^k \in \{0,1\} \\
\text{(1b)} & \forall T_k, \quad \sum_{P_q} y_q^k = 1 \\
\text{(1c)} & \forall F_{k,l}, \forall P_q \rightsquigarrow P_r, \quad x_{q,r}^{k,l} \leq y_q^k \\
\text{(1d)} & \forall T_l, \forall F_{k,l}, \forall P_r, \quad y_r^k + \sum_{P_q \rightsquigarrow P_r} x_{q,r}^{k,l} \geq y_r^l \\
\text{(1e)} & \forall P_q, \quad \sum_{T_k} y_q^k w_{q,k} \leq \tau \\
\text{(1f)} & \forall P_q \rightarrow P_r, \\
 & \qquad d_{q,r} = \sum_{\substack{P_s \rightsquigarrow P_t \text{ with} \\ P_q \rightarrow P_r \in P_s \rightsquigarrow P_t}} \; \sum_{F_{k,l}} x_{s,t}^{k,l} \text{data}_{k,l} \\
\text{(1g)} & \forall P_q \rightarrow P_r, \quad \frac{d_{q,r}}{\text{bw}_{q,r}} \leq \tau \\
\text{(1h)} & \forall P_q \quad \sum_{P_q \rightarrow P_r \in E_P} \frac{d_{q,r}}{B_q^{\text{out}}} \leq \tau \\
\text{(1i)} & \forall P_r \quad \sum_{P_q \rightarrow P_r \in E_P} \frac{d_{q,r}}{B_q^{\text{in}}} \leq \tau
\end{array}
\right. \tag{1}
$$

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. (The proof of this result is available in the research report [15].)

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

As the optimal linear programming approach can only be used on small problems, we now fallback to design heuristics.

4.2 Greedy mapping policies

In this section, we propose greedy strategies to find an allocation of task graphs on processors. Greedy algorithms are fast, easy to implement, and often effective to find reasonable solutions.

Simple greedy This heuristic is described by Algorithm 1. We first compute the weight of a task T_k as its maximum execution time over all processors. Then we consider the task with maximum weight, and allocate it to a processor so that the updated computation time of the processor is minimum. As we focus on steady state, we do not consider dependences and only try to minimize the processor occupation time. (Dependences are taken care of when building the schedule from the steady-state characterization [5].)

Algorithm 1: $\mathbf{Simple\_greedy}(G_P, G_A)$

```
foreach T_k do weight[T_k] ← max_{P_i} w_{i,k};
foreach P_i do processing_time[P_i] ← 0;
foreach T_k in decreasing order of weight do
    Find P_i such that processing_time[P_i] + w_{i,k} is
    minimized;
    processing_time[P_i] ←
    processing_time[P_i] + w_{i,k};
    mapping[T_k] ← P_i;
return mapping;
```

Refined greedy The previous heuristic attempts to balance the computing workload on processors, but do not take communications into account. We try to refine the search of an allocation in this heuristic, inspired from the classical HEFT algorithm for task graph scheduling [26]. Again, dependences are not taken into account since we focus on steady state (see the remark on SIMPLE_GREEDY). Algorithm 2 describes this heuristic.

Algorithm 2: $\mathbf{Refined\_greedy}(G_P, G_A)$

```
avg_comp_time[T_k] ← average computation time of
T_k over all processors;
avg_comm_time[F_{k,l}] ← average communication
time of F_{k,l} over all links;
foreach source T_k of the task graph do
    weight[T_k] ← 0;
foreach T_l in topological order do
    weight[T_l] ←
    max_{F_{k,l}} (weight[T_k] + avg_comm_time[F_{k,l}]) +
    avg_comp_time[T_l];
foreach T_k in decreasing order of weight do
    Find P_i such that the maximum of occupation
    time over all resources is minimized;
    mapping[T_k] ← P_i;
    Update the occupation time of all involved
    resources (P_i and communication links);
return mapping;
```

4.3 Rounding of the linear program

Since our problem is expressed as a mixed linear program, a natural way to find a solution is to relax the linear program to solve it over rational numbers, and then to round-off the rational solution into an integer one. Several different approaches exist for the rounding-off of rational solutions. We present two of them: a greedy rounding and a randomized one. Both variants are described in Algorithm 3.

Algorithm 3: $\mathbf{RLP}(G_P, G_A)$

```
Constraints ← initial set of constraints, given in
Linear Program (1);
for m = 1 to n do
    Solve over the rationals the linear program
    associated to Constraints;
    if using RLP_max then
        Find the maximum of the y_i^k's over all T_k's
        and P_i's, such that y_i^k has not yet been set;
    else if using RLP_rand then
        Randomly choose some task T_k which has not
        yet been mapped;
        Randomly choose a processor P_i, using
        probability y_j^k for P_j;
    Constraints ← Constraints ∪ {y_i^k = 1};
```

In the first variant (RLP_MAX), at each step, we search among the y_i^k's (which have not yet been set) for the one with the largest value. This y_i^k is then set to 1 in the Linear Program. After n steps, each task has been mapped to a processor, which defines a whole allocation.

In the second variant (RLP_RAND), we make use of a randomized rounding, which is known to sometimes lead to very efficient solutions [10]. At each step, we randomly select a task that has not yet been mapped. Then we randomly choose the processor that will process this task using a probability distribution defined by the y_i^k: each processor P_i has probability y_i^k to be chosen for the processing.

4.4 An involved strategy to delegate computations

In this Section, we present an iterative strategy to build an efficient allocation. Contrarily to the previous heuristics, this algorithm is not based on the linear program formulation. This method, called DELEGATE, consists in iteratively refining an allocation by moving some work from a highly loaded processor to a less loaded one. In the beginning, all tasks are mapped to the source processor P_{source}. Then, we select one task and some of its neighbors and *delegate* them to another processor. This refinement procedure is repeated

as long as the throughput can be improved, as described in Algorithm 4.

Algorithm 4: $\mathbf{DELEGATE}(G_P, G_A, depth)$

```
foreach T_k do current_mapping[T_k] ← P_source;
current_value ← evaluate(current_mapping);
continue ← TRUE;
while continue do
    best_value ← 0;
    foreach T_k do
        foreach P_i such that
        current_mapping[T_k] ≠ P_i do
            forall connected neighborhood S of T_k do
                mapping ← move
                (current_mapping, S, P_i);
                mapping ← refine_move
                (mapping, S, P_i);
                value ← evaluate(mapping);
                if (value > best_value) then
                    (best_value,best_mapping) ←
                                    (value,mapping);
    if (best_value > current_value) then
        (current_value,current_mapping) ←
                        (best_value,best_mapping);
        continue ← TRUE;
    else continue ← FALSE;
return current_mapping;
```

At each step, we consider a candidate move (T_k,P_i), i.e., delegating task T_k to processor P_i (assuming that T_k is not already mapped to P_i). Delegating a single task T_k to another processor may not be interesting because of communications involving this task. Thus, we look for a cluster of tasks containing T_k that it would be beneficial to delegate to P_i.

We define a neighborhood of T_k as 1) a connected set of tasks, 2) which contains T_k, and 3) which only contains tasks which are at most at a distance $depth$ from T_k in the task graph. $depth$ is a parameter of the algorithm. In practice we set the value of $depth$ to 2. We will see in Section 5 that this value is large enough for our purpose.

We test all neighborhoods, trying in turn to map each of them on processor P_i. This is done through the **move** function. We then select the best move among all neighborhoods of all tasks. If this best move induces an improvement in performance, we perform the move. Otherwise, we end the overall process.

Evaluation metric This algorithm strongly depends on the evaluation function, which is used both to identify the best move, and to decide whether the overall performance is improved. We have several possible choices for this evaluation function:

- Obviously, we could use the throughput of an allocation as a measure of its quality. We can compute the throughput as described in Section 3.3: the total throughput is the inverse of the maximum occupation time of any resource. It can similarly be computed with

$$\rho = \min\left\{\min_{P_q}\left\{\frac{1}{t_q^{\text{comp}}}, \frac{1}{t_q^{\text{out}}}, \frac{1}{t_q^{\text{in}}}\right\}, \min_{P_q \to P_r} \frac{1}{t_{q,r}}\right\}.$$

  Using the global throughput allows us to ensure that the overall performance is improved at each step, but may lead to sub-optimal scenarios: when two processors are evenly loaded, we can only decrease the occupation time of a single processor at each algorithm step. Two successive moves are thus required for the overall throughput to decrease. This cannot be done with this evaluation function the way we designed Algorithm 4.
- To overcome the issue of using the throughput metric, we rather use a different way to compare two allocations, thanks to the lexicographical order. Instead of computing a single value for each allocation, we sort all resource occupation times by decreasing order, and use the lexicographical order to compare two allocations. The underlying idea is to first minimize the occupation of the most used resource —the one defining the throughput— and then the occupation of the other resources.

Further improvements Algorithm 4 can be improved in several ways:

- To keep computation time low, we have to set parameters $depth$ to small values. However, this prevents large subset of tasks to be simultaneously delegated to a same processor. Thus, we introduce a function, **refine_move**, which enlarges the subset of tasks to delegate to a processor. It greedily considers in an arbitrary order any neighbor task of the tasks currently in the subset S, and add this task to S if this leads to a better allocation.
- The search for an allocation continues until Algorithm 4 cannot improve the allocation any further. When using the lexicographical order, however, the number of iterations might be large before no more local improvement is possible. To keep a low execution time, we could bound the maximum number of iterations.

In the following experiments, we always use the **refine_move** improvement, but never bound the number of iterations.

5 Performance evaluation

5.1 Reference heuristic

To assess the interest of using static-scheduling techniques, we compare our steady-state strategies to the classical scheduling algorithm HEFT [26]. Contrarily to the other heuristics, HEFT does not construct an allocation which is used for all instances of the task graph, but schedules all task graphs separately. Thus, HEFT can *a priori* use much more resources than our strategies, which uses at most one processor per task in the task graph.

5.2 Simulation settings

All algorithms are simulated using the SimGrid framework [7]. The number N of instances to process is set to a large number (between 100 and 1000). For all steady-state strategies (MLP, SIMPLE_GREEDY, REFINED_GREEDY, RLP_MAX, RLP_RAND, and DELEGATE), we run the corresponding algorithm to build the allocation; then we construct the schedule corresponding to the steady-state allocation for the N instances. For the reference heuristic, this schedule is directly constructed by applying HEFT to the collection of N instances. Then, each schedule is executed in the SimGrid simulator [7]. We compute the experimental throughput as the ratio between the total completion time and the number N of instances. For steady-state heuristics, we also compute a theoretical throughput, based on the study of Section 3.3. The theoretical and experimental throughputs may differ due to the slight differences between our multiport model and the SimGrid network model. Nevertheless, they are quite close in our experiments. Table 1 gives the average error (and its standard deviation) between theoretical and experimental throughputs for each algorithm. The more communication-aware the heuristics, the smaller the error.

We perform two sets of simulations. First, we compare all algorithms on rather small problems (up to 12 tasks in the tasks graphs); we have 135 platform/application scenarios in this set. Then, we compare the heuristics on larger simulation settings, with task graphs including up to 47 tasks; this set comprises 445 scenarios. We exclude the MLP algorithm from the latter set of problems because of its prohibitive running time on the larger task graphs.

Platforms We use several platforms modeling actual computing Grids. The descriptions of these platforms were obtained through the SimGrid simulator repository [7]:

- 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 gathering nearly 5000 cores,
- GridPP, the UK Grid infrastructure for particle physics.

Most of the time, users do not have access to a whole Grid but to a limited subset of a Grid, usually through a reservation. To simulate this behavior, a subset of the available processors is first randomly selected for each platform/application scenario, and then used by all heuristics. It is composed of around 10 processors for the small problems, and between 40 and 70 processors for larger problems.

Applications Several task graphs are used to assess the quality of our strategies, with between 8 and 12 tasks when the MLP algorithm is used, and up to 47 tasks otherwise:

(i) pipeAlign [23], a protein family analysis tool,
(ii) several random task graphs generated by the TGFF generator [12].

In order to evaluate the impact of communications on the quality of the result, we artificially modify the applications' communication-to-computation ratios (CCR) by multiplying the overall volume of communications by a constant factor. We use the CCR as the basis for our comparisons. There are many possible ways to define this ratio. We chose to define an average computation time t_{comp} by dividing the sum of all computation volumes by the average computational power of the platform. We similarly defined an average communication time t_{com} by dividing the sum of all communication volumes by the average bandwidth in the platform. We then define the CCR as the ratio $t_{\text{com}}/t_{\text{comp}}$.

Finally, we impose the first and last tasks of each task graph to be processed on the first processor P_{source}. P_{source} is then the processor used to communicate with the outside world: it receives the input data sets and sends back the results. These first and last tasks have a size 0 and correspond to the storage of input and output data. Our application settings includes both related and unrelated applications, as discussed in Section 3.1, but we do not distinguish them as they lead to comparable results.

5.3 Results

5.3.1 Study on small task graphs

In this set of experiments, we include all heuristics and the MLP algorithm which computes the optimal allocation. As different scenarios may lead to very different throughputs, we normalize all results so that the optimal single-allocation algorithm MLP has throughput one. An interesting question we want to answer is whether restricting to a single allocation limits the achievable throughput, compared to a strategy like HEFT. Top of Figure 3(a) shows that the optimal single-allocation strategy MLP is better than HEFT

MLP	SIMPLE_GREEDY	REFINED_GREEDY	RLP_MAX	RLP_RAND	DELEGATE
3% ($\pm$ 3%)	21% ($\pm$ 27%)	14% ($\pm$ 22%)	14% ($\pm$ 19%)	10% ($\pm$ 15%)	3% ($\pm$ 3%)

Table 1. Average error between theoretical and experimental throughputs for each algorithm.

as soon as communications are not negligible, that is when the CCR is greater than 0.01. When the communication-to-computation ratio exceeds 0.05, HEFT does not exceed 50% of the optimal throughput of a single allocation. This both justifies our claim that static scheduling techniques can outperform classical schedulers, and motivates the search for single-allocation schedules. When the communication-to-computation ratio is very small (smaller than 0.01), communications are negligible and the best solution may be to execute a different instance of the task graph on each processor (except for peculiar applications whose tasks have strong unrelated characteristics). This explains the performance of HEFT for very low values of the CCR: it is able to use more resources. On the other hand, when the CCR is very high (larger than one), sometimes communications are so expensive that all tasks must be mapped on the source processor P_{source}. All the heuristics that are able to detect this (DELEGATE and HEFT) then give the optimal throughput.

In the other diagrams of Figure 3(a), we compare the optimal allocation strategy MLP to the allocation-building heuristics described in Section 4. The DELEGATE heuristic always achieves the best throughput among the steady-state heuristics. Furthermore, its performance is very close to the optimal performance defined by the mixed linear program (MLP). We also notice that strategies based on the rounding-off of relaxed linear programs are not significantly more efficient than our greedy strategies, despite their higher complexity.

The attentive reader will notice that, sometimes, the DELEGATE heuristic builds an allocation with an higher throughput than the optimal allocation given by MLP. This apparent paradox does not contradict the optimality of MLP. Indeed, MLP gives an allocation with optimal *theoretical* throughput, and we have seen that the experimental throughput may slightly differ from the theoretical throughput. This is why DELEGATE is sometimes "better than the optimal".

5.3.2 Study on larger task graphs

Figure 3(b) shows a comparison between some steady-state heuristics (REFINED_GREEDY and DELEGATE) and the HEFT strategy on larger task graphs. Due to the high running times of the RLP_MAX and RLP_RAND heuristics, we only ran them on a subset of the larger task graphs on which they happened to perform poorly. Therefore, we do not report here on their performance.

As we cannot compute the optimal single-allocation throughput anymore, we normalize all results so that DELEGATE gives a throughput of one. Note that the normalized throughput is exactly the inverse of the normalized *makespan*, as we defined the practical throughput as the ratio between the number of task graph instances and the makespan.

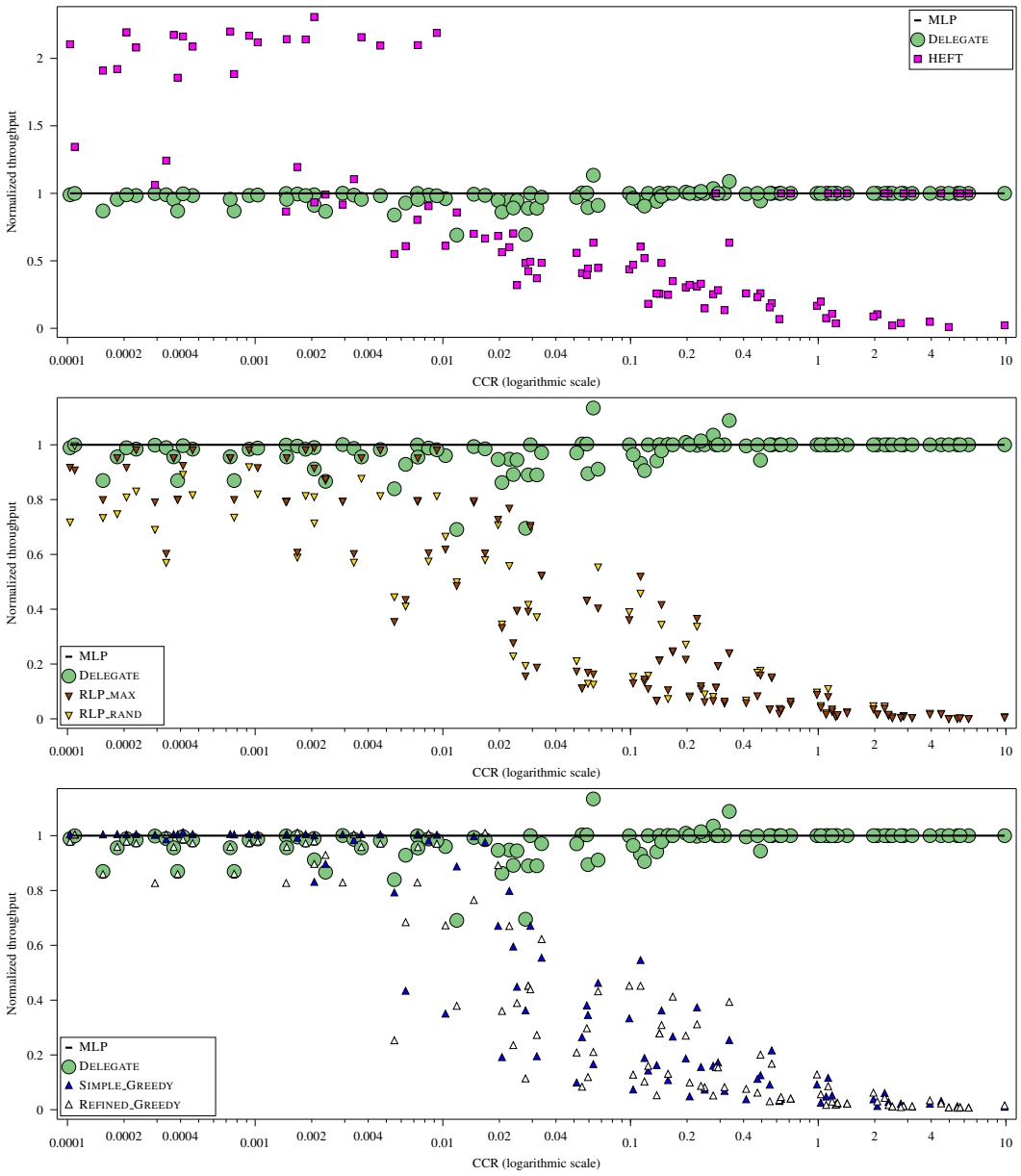
Greedy heuristics (SIMPLE_GREEDY and REFINED_GREEDY) performs similarly; we only plot one of them for readability. As for small task graphs, as soon as communications matter, DELEGATE provides the best results, while HEFT performs similarly to greedy strategies. However, when communications are almost negligible, greedy strategies and HEFT outperforms DELEGATE. In other words, DELEGATE is the best heuristics in the complex cases, that is when communications do not impose trivial solutions (no parallelism when communications are too large, one task graph per processor when there are no communications). On average, DELEGATE achieves makespan 2.05 times shorter than those of HEFT.

5.3.3 Running times

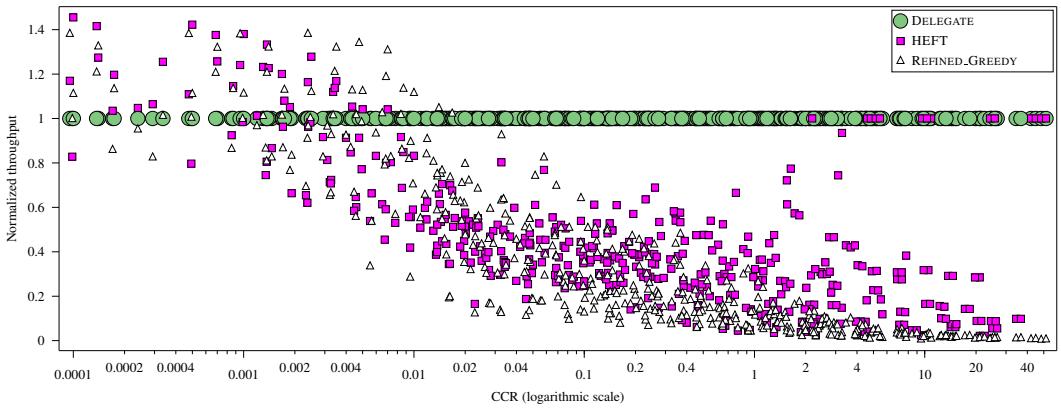
We also study the running times of the different heuristics. The CPLEX software [11] allows us to solve mixed linear programs in about one minute for the small settings. However, for task graphs larger than about 20 tasks, the running time is often more than a full day. Both RLP_MAX and RLP_RAND needs to solve many linear programs (over the rationals). Since these linear programs are quite big, the total running time of these heuristics often reach 10 minutes for the large settings. The greedy heuristics SIMPLE_GREEDY and REFINED_GREEDY are very fast (less than one second), whereas DELEGATE computes an allocation in about one half of the time needed for HEFT to compute its schedule on 1000 instances.

6 Conclusion and perspectives

In this paper, we have studied the scheduling of a collection of task graphs on a heterogeneous platform. Rather than attempting the classical makespan minimization, we have taken advantage of the regularity of our problem to optimize the system throughput by applying steady-state techniques. We have presented a mixed linear programming approach to compute the optimal allocation, and several heuristic algorithms to build practical solutions. We



(a) Performance on the small task graphs. Results are normalized such that MLP has throughput one.



(b) Performance on the larger task graphs. Results are normalized such that DELEGATE has throughput one.

Figure 3. Simulation results. X-axis is the communication-to-computation ratio.

have performed extensive simulations to compare the performance of our heuristics to a classical scheduler, HEFT. Simulation results show the benefit of our approach as soon as communication times are not negligible. Our heuristic of choice, DELEGATE, almost always gives the best makespan. On average, DELEGATE achieves makespans which are twice shorter than HEFT's ones, in our simulation settings, while having a lower running time.

References

[1] I. Ahmad and Y.-K. Kwok. On exploiting task duplication in parallel program scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 9(9):872–892, 1998.

[2] 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.

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

[4] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Pipelining broadcasts on heterogeneous platforms. *IEEE Trans. Parallel Distributed Systems*, 16(4):300–313, 2005.

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

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

[7] H. Casanova, A. Legrand, and M. Quinson. SimGrid: a generic framework for large-scale distributed experimentations. In *Proceedings of IEEE UKSIM/EUROSIM'08*, 2008.

[8] 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.

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

[10] D. Coudert, H. Rivano, and X. Roche. A combinatorial approximation algorithm for the multicommodity flow problem. In *WAOA*, pages 256–259, 2003.

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

[12] R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: task graphs for free. In *CODES*, pages 97–101, 1998.

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

[14] M. Gallet, L. Marchal, and F. Vivien. Allocating series of workflows on computing grids. In *ICPADS*. IEEE Computer Society, 2008.

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

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

[17] 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.

[18] T. Glatard, J. Montagnat, D. Lingrand, and X. Pennec. Flexible and efficient workflow deployement of data-intensive applications on grids with MOTEUR. *Int. Journal of High Performance Computing and Applications*, 2008.

[19] B. Hong and V. Prasanna. Distributed adaptive task allocation in heterogeneous computing environments to maximize throughput. In *IPDPS'2004*. IEEE CS Press, 2004.

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

[21] 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.

[22] M. A. Palis, J.-C. Liou, and D. S. L. Wei. Task clustering and scheduling for distributed memory parallel architectures. *IEEE Trans. Parallel Distrib. Syst.*, 7(1):46–55, 1996.

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

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

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

[26] 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.

[27] V. Volkov and J. Demmel. LU, QR and Cholesky factorizations using vector capabilities of GPUs. Technical Report UCB/EECS-2008-49, U. of California, Berkeley, May 2008.

[28] 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.