Workload balancing and throughput optimization for heterogeneous systems subject to failures

Anne Benoit — Alexandru Dobrila — Jean-Marc Nicod — Laurent Philippe

N° 7532

February 2011

Distributed and High Performance Computing

ISSN 0249-6399 ISRN INRIA/RR--7532--FR+ENG

Workload balancing and throughput optimization for heterogeneous systems subject to failures

Anne Benoit , Alexandru Dobrila , Jean-Marc Nicod , Laurent Philippe

Theme : Distributed and High Performance Computing
Équipe-Projet GRAAL

Rapport de recherche n° 7532 — February 2011 — 22 pages

Abstract: In this report, we study the problem of optimizing the throughput of applications for heterogeneous platforms subject to failures. The considered applications are composed of a sequence of consecutive tasks linked as a linear graph (pipeline), with a type associated to each task. The challenge is to specialize the machines of a target platform to process only one task type, given that every machine is able to process all the types before being specialized, to avoid costly context or setup changes. Each instance can thus be performed by any machine specialized in its type and the workload of the system can be shared among a set of specialized machines. For identical machines, we prove that an optimal solution can be computed in polynomial time. However, the problem becomes NP-hard when two machines can compute the same task type at different speeds. Several polynomial time heuristics are presented for the most realistic specialized settings. Experimental results show that the best heuristics obtain a good throughput, much better than the throughput obtained with a random mapping, and close to the optimal throughput in the particular cases on which the optimal throughput can be computed.

Key-words: workload balancing, distributed systems, fault tolerance, scheduling, optimization heuristics.

Équilibrage de charge et optimisation du débit dans les systèmes hétérogènes sujets aux pannes

Résumé : Dans ce rapport, nous étudions le problème de l'optimisation du débit d'applications de type pipeline dans un environnement hétérogène sujet à des pannes. Les applications sont constituées d'un ensemble de tâches consécutives typées et organisées selon une chaîne linéaire ou pipeline. Le but est ici de spécialiser les machines de la plate-forme d'exécution afin qu'elles ne traitent qu'un seul type de tâches, sachant qu'au départ elles peuvent exécuter tous les types. Cela permet d'éviter des reconfigurations coûteuses entre tâches de types différents sur une même machine. Ainsi, les instances d'une même tâche peuvent être distribuées sur plusieurs machines spécialisées pour le type de cette tâche, ce qui permet une répartition de la charge du système sur un ensemble de machines spécialisées. Lorsque la plate-forme est composée de machines identiques, nous prouvons qu'une solution optimale peut être trouvée en temps polynomial. Par contre, le problème devient NP-complet dès lors que deux machines peuvent traiter une même tâche à des vitesses différentes. Ce faisant, plusieurs heuristiques sont présentées dans le cas le plus réaliste d'un système spécialisé. Les résultats expérimentaux montrent que les meilleures heuristiques obtiennent de bons résultats en terme de débit, meilleurs qu'avec une allocation aléatoire, et que les débits atteints sont très proches des débits optimaux dans les cas particuliers pour lesquels une solution optimale peut être calculée.

Mots-clés : équilibrage de charge, systèmes distribués, tolérance aux pannes, ordonnancement, heuristiques d'optimisation.

Contents

1 Introduction

Most of the distributed environments are subject to failures, and each component of the environment has its own failure rate. Assuming that a fault may be tolerated, as for instance in asynchronous systems [1] or production systems, the failures have an impact on the system performance. When scheduling an application onto such a system, either we can account for failures to help improve the performance in case of failures, or ignore them. In some environments, such as computing grids, this failure rate is so high that we cannot ignore failures when scheduling applications that last for a long time as a batch of input data processed by pipelined tasks for instance. This is also the case for micro-factories where a production is composed of several instances of the same micro-component that must be processed by cells.

In this paper, we deal with scheduling and mapping strategies for *coarse-grain* workflow applications [18, 19], for which each task of the workflow is of a given type, and subject to failures. The considered resources are typically execution resources grouped in a distributed platform, a grid or a micro-factory, on which we schedule either a batch of input data processed by pipelined tasks or a production (streaming application). Each resource provides functions or services with heterogeneous fault rates and efficiencies. In a streaming application, a series of jobs enters the workflow and progresses from task to task until the final result is computed. Once a task is mapped onto a set of dedicated resources (known in the literature as *multi-processor tasks* [2, 7]), the computation requirements and the failure rates for each resource when processing one job are known. After an initialization delay, a new job is completed every period and it exits the workflow. The period is therefore defined as the longest cycle-time of a resource, and it is the inverse of the throughput that can be achieved. We target coarse-grain applications and platforms on which the cost of communications is negligible in comparison to the cost of computations.

In the distributed computing system context, a use case of a streaming application is for instance an image processing application where images are processed in batches, on a SaaS (Software as a service) platform. In this context, failures may occur because of the nodes, but they also may be impacted by the complexity of the service [9]. On the production side, a use case is a micro-factory [13, 5, 12] composed of several cells that provides functions as assembly or machining. But, at this scale, the physical constraints are not totally controlled and it is mandatory to take faults into account in the automated command. A common property of these systems is that we cannot use replication, as for instance in [3, 14, 10], to overcome the faults. For streaming applications, it may impact the throughput to replicate each task. For a production which deals with physical objects, replication is not possible. Fortunately, losing a few jobs may not be a big deal; for instance, the loss of some images in a stream will not alter the result, as far as the throughput is maintained, and losing some micro-products is barely more costly than the occupation of the processing resources that have been dedicated to it.

The failure model is based on the Window-Constrained [15] model, often used in real-time environment. In this model, only a fraction of the messages will reach their destination. The losses are not considered as a failure but as a guarantee: for a given network, a Window-Constrained scheduling [17, 16] can guarantee that no more than x messages will be lost for every y sent messages.

The paper is organized as follows. Section 2 presents the framework and formalizes the optimization problems tackled in the paper. An exhaustive complexity study is provided in Section 3: we exhibit some particular polynomial problem instances, and prove that the remaining problem instances are NP-hard problems. In Section 4, we design a set of polynomial-time heuristics to solve the most general problem instance, building upon complexity results, and in particular linear program formulations to solve sub-problems. Moreover, we conduct extensive simulations to assess the relative and absolute performance of the heuristics. Finally, we conclude in Section 5.

2 Framework and optimization problems

In this section, we define the problems that we tackle. First we present the application and platform models. Then, we discuss the objective function and the rules of the game, before introducing formally the optimization problems.

2.1 Applicative framework

The application model is a stream of job instances that are executed one after another on the platform. All the job instances have the same structure, a workflow, but different inputs. The workflow is composed of a set $\mathcal{N}$ of n tasks: $\mathcal{N} = \{T_1, T_2, \dots, T_n\}$ that are linked by precedence constraints. From the tasks point of view, the application model may also be seen as a pipeline where the tasks are successively applied to the job instances and the workflow can be modeled by a linear chain in which the vertices are tasks, and edges represent dependencies between tasks. An example of application tasks is represented on Figure 1.

In the workflow, we note x_i the average number of jobs processed by task T_i to ouput one job out of the system, x_{in} the number of jobs that input the platform and x_{out} the number of jobs that output the platform. Here, we want to maximize the throughput and, as some job losses may occur because of failures, we need to input more jobs in the platform than what we get out of the system ($x_{in} > x_{out}$). Note that x_{i+1} depends on x_i and of the properties, failures and computing power, of the machines that process T_i.

A type is associated to each task as the same operation may be applied several times to the same job. Thus, we have a set $\mathcal{T}$ of p task types with $n \geq p$ and a function $t : \{1, \dots, n\} \to \mathcal{T}$ which returns the type of a task: $t(i)$ is the type of task T_i, for $1 \leq i \leq n$.

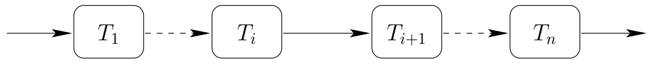

Figure 1: Example of application.

2.2 Target platform

The target platform is distributed and heterogeneous. It consists of a set $\mathcal{M}$ of m machines (a cell in the micro-factory or a host in a grid platform): $\mathcal{M} = \{M_1, M_2, \dots, M_m\}$.

The task processing time depends on the machine that performs it: it takes $w_{i,u}$ units of time to machine M_u to execute task T_i on one job. Each machine is able to process all the task types. But, to avoid costly context or setup changes during execution, the machines may be specialized to process only one task type. Moreover, the machine are interconnected by a complete graph but we do not take communication times into account as we consider that the processing time is much greater than the communication time (coarse-grain applications).

An additional characteristic of our framework is that failures may occur. It may happen that a job (or data, or product) is lost (or damaged) while a task is being executed on this job instance. For instance, an electrostatic charge may be accumulated on an actuator and a the piece will be pushed away rather than caught or a message will be lost due to network contention. Due to our application setting, we deal only with transient failures, as defined in [8]. The tasks are failing for some of the job instances, but we do not consider a permanent failure of the machine responsible of the task, as this would lead to a failure for all the remaining jobs to be processed and the inability to finish them. As explained above, in order to deal with failures, we process more inputs than needed, so that at the end, the required throughput is reached.

The failure rate of task T_i performed onto machine M_u is the percentage of failure for this task and is denoted $f_{i,u} = \frac{l_{i,u}}{b_{i,u}}$, where $l_{i,u}$ is the number of lost products each time $b_{i,u}$ products have been processed ($l_{i,u} \leq b_{i,u}$).

2.3 Objective function

Our goal is to assign tasks to machines so as to optimize some key performance criteria. A task can be allocated to several machines, and $q(i,u)$ is the quantity of task T_i executed by machine M_u; if $q(i,u) = 0$, T_i is not assigned to M_u.

Recall that x_i is the average number of jobs processed by task T_i to output one job out of the system. We must have, for each task T_i, $\sum_{u=1}^{m} q(i,u) = x_i$, i.e., enough jobs are processed for task T_i in the system.

In our framework, several objective functions could be optimized. For instance, one may want to produce a mapping of the tasks on the machines as reliable as possible, i.e., minimize the number x_{in} of products to input in the system. Rather, we consider that losing one instance is not a big deal, and we focus on a performance criteria, the throughput. The goal is to maximize the number of instances processed per time unit, making abstraction of the initialization and clean-up phases. This objective is important when a large number of instances must be processed. Actually, rather than maximizing the throughput of the application, we deal with the equivalent optimization problem that minimize the *period*, the inverse of the throughput.

Unfortunately, we cannot compute the number x_i of jobs needed by task T_i, before allocating that task to some machines. Remember that each task T_i has an unique successor T_{i+1} and that x_{i+1} is the amount of jobs needed by T_{i+1} as input. Since T_i is distributed on several machines with different failure rates, we have $\sum_{u=1}^{m} (q(i,u) \times (1 - f_{i,u})) = x_{i+1}$, where $q(i,u) \times (1 - f_{i,u})$ represents the amount of jobs output by the machine M_u if $q(i,u)$ jobs are treated by that machine. For each task, we sum all the instances treated by all the machines.

We are now ready to define the cycle-time ct_u of machine M_u: it is the time needed by M_u to execute all the tasks T_i with $q(i,u) > 0$: $ct_u = \sum_{i=1}^{n} q(i,u) \times$

$w_{i,u}$. The objective function is to minimize the maximum cycle-time, which corresponds to the period of the system: $\min \max_{1 \leq u \leq m} ct_u$.

2.4 Rules of the game

Different rules of the game may be enforced to define the allocation, i.e., the $q(i,u)$ values. For *one-to-many* mappings, we enforce that a single task must be mapped onto each machine: $\forall 1 \leq i, i' \leq n, \quad i \neq i', \quad q(i,u) > 0 \Rightarrow q(i',u) = 0$. For instance, on Figure 2, we have an application graph (a) that must be mapped on a platform graph (b). The result is shown in (c) where we can see that one machine can handle only one task. This mapping is quite restrictive because we must have at least as many machines as tasks. Note that a task can be split in several instances and each instance is executed by different machines.

We relax this rule to allow for *specialized* mappings, in which several tasks of the same type can be mapped onto the same machine: $\forall 1 \leq i, i' \leq n \; s.t. \; t(i) \neq t(i'), q(i,u) > 0 \Rightarrow q(i',u) = 0$. For instance, on Figure 3, we have five tasks with types $t(1) = t(3) = t(5) = 1$ and $t(2) = t(4) = 2$. Machine M_3 computes task T_1, therefore it can also execute T_3 and T_5 but not T_2 and T_4. As types are not dedicated to machines, T_5 can also be assigned to another machine, for instance M_1. Note that if each task has a different type, the specialized mapping and the one-to-many mapping are equivalent.

Finally, *general* mappings have no constraints: any task (no matter the type) can be mapped on any machine, as illustrated on Figure 4.

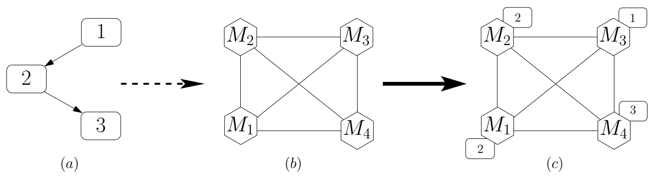

Figure 2: One-to-many mapping.

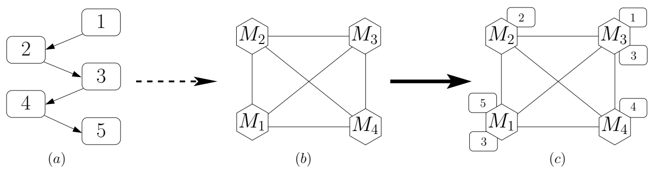

Figure 3: Specialized mapping. t(1)=t(3)=t(5)=1 and t(2)=t(4)=2.

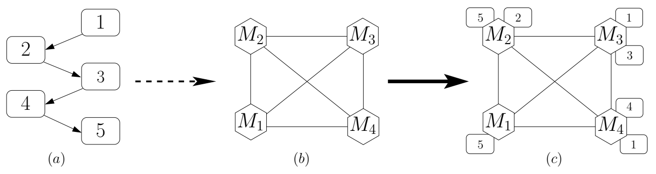

Figure 4: General mapping. t(1)=t(3)=1, t(2)=t(4)=2 and t(5) = 3.

2.5 Problem definition

For the optimization problem that we consider, the three important parameters are: (i) the rules of the game (*one-to-many (o2m)* or *specialized (spe)* or *general (gen)* mapping); (ii) the failure model (f if failures are all identical, f_i if the failure for a same task is identical on two different machines, f_u if the failure rate depends only on the machine, and the general case $f_{i,u}$); and (iii) the computing time (w if the processing times are all identical, w_i if it differs only from one task to another, w_u if it depends only on the machine, and $w_{i,u}$ in the general case). We are now ready to formally define the optimization problem:

Definition 1 (MinPer(R, F, W))**.** *Given an application and a target platform, with a failure model $F = \{f|f_i|f_u|f_{i,u}|*\}$ and computation times $W = \{w|w_i|w_u|w_{i,u}|*\}$, find a mapping, i.e., values of $q(i,u)$ such that for each task T_i ($1 \leq i \leq n$), $\sum_{u=1}^{m} q(i,u) = x_i$, following rule $R = \{o2m|spe|gen|*\}$ which minimizes the period of the application,* $\max_{1 \leq u \leq m} \sum_{i=1^n} q(i,u) \times w_{i,u}$.

For instance, MinPer$(o2m, f_i, w_i)$ is the problem of minimizing the period with a *one-to-many* mapping, where both failure rates and execution time depend only on the tasks. Note that $*$ is used to express the problem with any variant of the corresponding parameter; for instance, MinPer$(*, f_{i,u}, w)$ is the problem of minimizing the period with any mapping rule, where failure rates are general, while execution times are all identical.

3 Complexity results

In this section, we assess the complexity of the different instances of the problem that we named the MinPer(R, F, W) problem. First we establish some properties on the solution of the problems with $F = f_i$ in Section 3.1. Then, building upon these properties, we provide the complexity of these problems in Section 3.2. Finally, we discuss the most general problems with $F = f_{i,u}$ in Section 3.3.

Even though the general problem is NP-hard, we show in Section 3.4 that once the allocation of tasks to machines is known, we can optimally decide how to share tasks between machines, in polynomial time. Also, we give an integer linear program to solve the problem (in exponential time) in Section 3.5.

3.1 Particularities of the MinPer$(*, f_i, *)$ problems

In this section, we focus on the MinPer$(*, f_i, *)$ problems, and we first show how these problems can be simplified. Indeed, in this case, the number of products that should be computed for task T_i at each period, x_i, is independent of the allocation of tasks to machines. We can therefore ignore the failure probabilities, and focus on the computation of the period of the application.

The following Lemma 1 allows us to further simplify the problem: tasks of similar type can be grouped and processed as a single *equivalent* task.

Lemma 1. *For* MinPer$(*, f_i, w_i)$ *or* MinPer$(*, f_i, w_u)$*, there exists an optimal solution in which all tasks of the same type are executed onto the same set of* *machines, in equal proportions:*

$$\forall 1 \leq i, j \leq n \text{ with } t(i) = t(j), \\ \exists\, \alpha_{i,j} \in \mathbb{Q} \text{ such that } \forall 1 \leq u \leq m,\ q(i,u) = \alpha_{i,j} \times q(j,u)\ . \tag{1}$$

Proof. Let OPT be an optimal solution to the problem, of period P. Let t be a task type, and, without loss of generality, let $T_1, \ldots, T_k$ be the k tasks of type t, with $k \leq n$, and let $M_1, \ldots, M_v$ be the set of machines specialized to type t in the optimal solution OPT (i.e., they have been allocated only tasks from $T_1, \ldots, T_k$), with $v \leq m$. In the optimal solution, $q(i,u)$ is the proportion of task T_i assigned to machine M_u. For each task T_i, $1 \leq i \leq k$, we have $\sum_{1\leq u\leq v} q(i,u) = x_i$, and for each machine M_u, $1 \leq u \leq v$, we have $\sum_{1\leq i\leq k} q(i,u) w_{i,u} \leq P$, where $w_{i,u}$ may be either w_i or w_u, depending upon the problem instance.

We build a new optimal solution, OPT', which follows Equation (1). The proportion of task T_i assigned to machine M_u in this solution is $q'(i,u)$, and $x^* = \sum_{1\leq i\leq k} x_i$.

Let us start with the MinPer$(*, f_i, w_u)$ problem. We define $q'(i,u) = \frac{x_i}{x^*} \times \frac{P}{w_u}$. For task T_i, $\sum_{1\leq u\leq v} q'(i,u) \geq \frac{x_i}{x^*} \times \sum_{1\leq u\leq v} \sum_{1\leq i\leq k} q(i,u)$, by definition of OPT, and therefore $\sum_{1\leq u\leq v} q'(i,u) \geq x_i$: we have distributed all the work for this task. Moreover, by construction, $\sum_{1\leq i\leq k} q'(i,u) w_u = P$: solution OPT' is optimal (its period is P). We have built an optimal solution which satisfies Equation (1), with $\alpha_{i,j} = \frac{x_i}{x_j}$, by redistributing the work of each task on each machine in equal proportion.

The reasoning is similar for the MinPer$(*, f_i, w_i)$ problem, except that w_i is now depending on the task, and therefore we define $q'(i,u) = \frac{x_i}{w_i x^*} \times P$. We still have the property that all the work is distributed, and the period of each machine is still P. The only difference relies in the values of $\alpha_{i,j}$, which are now also depending upon the w_i: $\alpha_{i,j} = \frac{x_i}{x_j} \times \frac{w_j}{w_i}$.

For each problem instance, we have built an optimal solution which follows Equation (1), therefore concluding the proof. □

Corollary 1. *For* MinPer$(*, f_i, w_i)$ *or* MinPer$(*, f_i, w_u)$*, we can group all tasks of same type t as a single* equivalent *task $T_t^{(eq)}$ such that*

$$x_t^{(eq)} = \sum_{1\leq i\leq n | t(i)=t} x_i\ .$$

Then, we can solve this problem with the one-to-many rule, and deduce the solution of the initial problem.

Proof. Following Lemma 1, we search for the optimal solution which follows Equation (1). Since all tasks of the same type are executed onto the same set of machines in equal proportions, we can group them as a single equivalent task. The amount of work to be done by the set of machines corresponds to the total amount of work of the initial tasks, i.e., for a type t, $\sum_{1\leq i\leq n|t(i)=t} x_i$.

The one-to-many rule decides on which set of machines each equivalent task is mapped, and then we share the initial tasks in equal proportions to obtain the solution to the initial problem: if task T_i is not mapped on machine M_u, then $q(i,u) = 0$, otherwise

$$q(i,u) = \frac{x_i}{x_{t(i)}^{(eq)}} \times \frac{P}{w_{i|u}}\ ,$$

where $w_{i|u} = \{w_i \mid w_u\}$, depending upon the problem instance. □

3.2 Complexity of the MinPer$(*, f_i, *)$ problems

We are now ready to establish the complexity of the MinPer$(*, f_i, *)$ problems. Recall that n is the number of tasks, m is the number of machines, and p is the number of types.

Results are summarized in Table 1. The MinPer$(*, f_i, w_i)$ problems can all be solved in polynomial time, as well as MinPer(gen, f_i, w_u), while this latter problem with w_u becomes NP-hard for one-to-many and specialized mappings. For the most general case of $w_{i,u}$, one-to-many and specialized mappings are NP-hard since they were NP-hard with w_u, and a linear program allows us to solve MinPer$(gen, f_i, w_{i,u})$, which remains polynomial.

We start by providing polynomial algorithms for one-to-many and specialized mappings with w_i (Theorem 1 and Corollary 2). Then, we discuss the case of general mappings, which can also be solved in polynomial time (Theorem 2). Finally, we tackle the instances which are NP-hard (Theorem 3).

Theorem 1. MinPer$(o2m, f_i, w_i)$ *can be solved in polynomial time* $O(m \times \log n)$*.*

Proof. First, note that solving this one-to-many problem amounts to decide on how many machines each task is executed (since machines are identical), and then split the work evenly between these machines to minimize the period. Hence, if T_i is executed on k machines, $q(i, u) = \frac{x_i}{k}$, where M_u is one of these k machines, and the corresponding period is $\frac{x_i}{k} \times w_i$.

We exhibit a dynamic programming algorithm which computes the optimal solution in polynomial time. We compute $P(i, k)$, which is the period that can be achieved to process tasks $T_1, \dots, T_i$ with k machines. The solution to the problem is $P(n, m)$, and the recurrence writes:

$$P(i,k) = \min_{1 \le k' \le k} \left(\max \left(P(i-1, k-k'), \frac{x_i}{k'} \times w_i \right) \right) ,$$

with the initialization $P(1, k) = \frac{x_i}{k} \times w_i$ (we use all remaining machines to process the last task), and $P(i, 0) = +\infty$ (no solution if there are still some tasks to process but no machine left). There are $n \times m$ values of $P(i, k)$ to compute, and the computation takes a time in $O(m)$. Therefore, the complexity of this algorithm is of order $O(n \times m^2)$.

Note that it is also possible to solve the problem greedily. The idea is to assign initially one machine per task (note that there is a solution only if $m \ge n$), sort the tasks by non-increasing period, and then iteratively add a machine to the task whose machine(s) have the greater period, while there are some machines available. Let g_i be the current number of machines assigned to task T_i: the corresponding period is $\frac{x_i}{g_i} \times w_i$. At each step, we insert the task whose period has been modified in the ordered list of tasks, which can be done in $O(\log n)$ (binary search). The initialization takes a time $O(n \log n)$ (sorting the tasks), and then there are $m - n$ steps of time $O(\log n)$. Since we assume $m \ge n$, the complexity of this algorithm is in $O(m \times \log n)$. To prove that this algorithm returns the optimal solution, let us assume that there is an optimal solution of period P_{opt} that has assigned o_i machines to task T_i, while the greedy algorithm has assigned

g_i machines to this same task, and its period is $P_{greedy} > P_{opt}$. Let T_i be the task which enforces the period in the greedy solution (i.e., $P_{greedy} = x_i w_i / g_i$). The optimal solution must have given at least one more machine to this task, i.e., $o_i > g_i$, since its period is lower. This means that there is a task T_j such that $o_j < g_j$, since $\sum_{1 \le i \le n} o_i \le \sum_{1 \le i \le n} g_i = m$ (all machines are assigned with the greedy algorithm). Then, note that since $o_j < g_j$, because of the greedy choice, $x_j w_j / o_j \ge x_i w_i / g_i$ (otherwise, the greedy algorithm would have given one more machine to task T_i). Finally, $P_{opt} \ge x_j w_j / o_j \ge x_i w_i / g_i = P_{greedy}$, which leads to a contradiction, and concludes the proof. □

Corollary 2. *MinPer(spe, f_i, w_i) can be solved in polynomial time $O(n + m \times \log p)$.*

Proof. For the specialized mapping rule, we use Corollary 1 to solve the problem: first we group the n tasks by types, therefore obtaining p equivalent tasks, in time $O(n)$. Then, we use Theorem 1 to solve the problem with p tasks, in time $O(m \times \log p)$. Finally, the computation of the mapping with equal proportions is done in $O(n)$, which concludes the proof. □

Theorem 2. *MinPer$(gen, f_i, *)$ can be solved in polynomial time.*

Proof. We exhibit a linear program to solve the problem for the general case with $w_{i,u}$. Note however that the problem is trivial for w_i or w_u: we can use Corollary 1 to group all tasks as a single equivalent task, and then share the work between machines as explained in the corollary.

In the general case, we solve the following (rational) linear program, where the variables are P (the period), and $q(i,u)$, for $1 \le i \le n$ and $1 \le u \le m$.

$$\begin{array}{ll} & \text{Minimize } P \\ & \text{subject to} \\ \text{(i)} & q(i,u) \ge 0 \text{ for } 1 \le i \le n, 1 \le u \le m \\ \text{(ii)} & \sum_{1 \le u \le m} q(i,u) = x_i \text{ for each task } T_i \text{ with } 1 \le i \le n \\ \text{(iii)} & \sum_{1 \le i \le n} q(i,u) \times w_{i,u} \le P \text{ for each machine } M_u \text{ with } 1 \le u \le m \end{array} \quad (2)$$

The size of this linear program is clearly polynomial in the size of the instance, all $n \times m + 1$ variables are rational, and therefore it can be solved in polynomial time [11]. □

Finally, we prove that the remaining problem instances are NP-hard (one-to-many or specialized mappings, with w_u or $w_{i,u}$). Since MinPer$(o2m, f_i, w_u)$ is a special case of all other instances, it is sufficient to prove the NP-completeness of the latter problem.

Theorem 3. *The MinPer$(o2m, f_i, w_u)$ problem is NP-hard in the strong sense.*

Proof. We consider the following decision problem: given a period P, is there a one-to-many mapping whose period does not exceed P? The problem is obviously in NP: given a period and a mapping, it is easy to check in polynomial time whether it is valid or not. The NP-completeness is obtained by reduction from 3-PARTITION [6], which is NP-complete in the strong sense.

We consider an instance $\mathcal{I}_1$ of 3-PARTITION: given an integer B and $3n$ positive integers $a_1, a_2, \ldots, a_{3n}$ such that for all $i \in \{1, \ldots, 3n\}$, $B/4 < a_i < B/2$

	$\boxed{f_i}$		$\boxed{f_{i,u}}$	
	w_i	w_u or $w_{i,u}$	w_i	w_u or $w_{i,u}$
o2m or spe	polynomial	NP-hard	NP-hard	NP-hard
gen	polynomial	polynomial	polynomial	polynomial

Table 1: Complexity of the MinPer problems.

and with $\sum_{i=1}^{n} a_i = nB$, does there exist a partition $I_1, \dots, I_n$ of $\{1, \dots, 3n\}$ such that for all $j \in \{1, \dots, n\}$, $|I_j| = 3$ and $\sum_{i \in I_j} a_i = B$? We build the following instance $\mathcal{I}_2$ of our problem with n tasks, such that $x_i = B$, and $m = 3n$ machines with $w_u = 1/a_u$. The period is fixed to $P = 1$. Clearly, the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$. We now show that $\mathcal{I}_1$ has a solution if and only if $\mathcal{I}_2$ does.

Suppose first that $\mathcal{I}_1$ has a solution. For $1 \le i \le n$, we assign task T_i to the machines of I_i: $q(i,u) = a_u$ for $u \in I_i$, and $q(i,u) = 0$ otherwise. Then, we have $\sum_{1 \le u \le m} q(i,u) = \sum_{u \in I_i} a_u = B$, and therefore all the work for task T_i is done. The period of machine M_u is $\sum_{1 \le i \le n} q(i,u) \times w_u = a_u/a_u = 1$, and therefore the period of 1 is respected. We have a solution to $\mathcal{I}_2$.

Suppose now that $\mathcal{I}_2$ has a solution. Task T_i is assigned to a set of machines, say I_i, such that $\sum_{u \in I_i} q(i,u) = B$, and $q(i,u) \le a_u$ for all $u \in I_i$. Since all the work must be done, by summing over all tasks, we obtain $q(i,u) = a_u$, and the solution is a 3-partition, which concludes the proof. □

3.3 Complexity of the MinPer$(*, f_{i,u}, *)$ problems

When we consider problems with $f_{i,u}$ instead of f_i, we do not know in advance the number of jobs to be computed by each task in order to have one job exiting the system, since it depends upon the machine on which the task is processed. However, we are still able to solve the problem with general mappings, as explained in Theorem 4. For one-to-many and specialized mappings, the problem is NP-hard with w_u, since it was already NP-hard with f_i in this case (see Theorem 3). We prove that the problem becomes NP-hard with w_i in Theorem 5, which illustrates the additional complexity of dealing with $f_{i,u}$ rather than f_i. Results are summarized in Table 1.

Theorem 4. MinPer$(gen, f_{i,u}, *)$ *can be solved in polynomial time.*

Proof. We modify the linear program (2) of Theorem 2 to solve the case with general failure rates $f_{i,u}$. Indeed, constraint (ii) is no longer valid, since the x_i are not defined before the mapping has been decided. It is rather replaced by constraints (iia) and (iib):

$$\text{(iia)} \sum_{1 \le u \le m} q(n,u) \times (1 - f_{n,u}) = 1 \;;$$

$$\text{(iib)} \sum_{1 \le u \le m} q(i,u) \times (1 - f_{i,u}) = \sum_{1 \le u \le m} q(i+1,u) \text{ for each } T_i \text{ with } 1 \le i < n \,.$$

Constraint (iia) states that the final task must output one job, while constraint (iib) expresses the number of jobs that should be processed for task T_i, as a function of the number for task T_{i+1}. There are still $n \times m + 1$ variables which are rational, and the number of constraints remains polynomial, therefore this linear program can be solved in polynomial time [11]. $\square$

Theorem 5. *The* MinPer$(o2m, f_{i,u}, w_i)$ *problem is NP-hard.*

Proof. We consider the following decision problem: given a period P, is there a one-to-many mapping whose period does not exceed P? The problem is obviously in NP: given a period and a mapping, it is easy to check in polynomial time whether it is valid or not. The NP-completeness is obtained by reduction from SUBSET-PRODUCT-EQ (SPE), which is NP-complete (trivial reduction from SUBSET-PRODUCT [6]).

We consider an instance $\mathcal{I}_1$ of SPE: given an integer B, and $2n$ positive integers $a_1, a_2, \dots, a_{2n}$, does there exist a subset I of $\{1, \dots, 2n\}$ such that $|I| = n$ and $\prod_{i \in I} a_i = B$? Let $C = \prod_{1 \leq i \leq 2n} a_i$.

We build the following instance $\mathcal{I}_2$ of our problem with $2n + 2$ tasks, and $2n + 2$ machines, so that the mapping has to be a one-to-one mapping (i.e., one task per machine). We ask whether we can obtain a period $P = 1$. Tasks T_1 and T_{n+2} are such that they should be allocated to machines M_{2n+1} and M_{2n+2}: the other machines never successfully compute a job for these tasks. We have:

- $w_1 = B/C^2$ and $w_{n+2} = 1/B$;
- $f_{1,2n+1} = f_{n+2,2n+2} = 0$ for $1 \leq u \leq 2n + 2$ (i.e., no failures in this case);
- $f_{1,u} = f_{n+2,v} = 1$ for $u \neq 2n + 1$ and $v \neq 2n + 2$ (i.e., total failure in this case).

The values of the failure probabilities mean that machine M_{2n+1} (resp. M_{2n+2}) never fails on task T_1 (resp. T_{n+2}), while all the other machines always fail on these tasks, i.e., the number of failed product is equal to the number of products entering the machine. No final product is output if these tasks are mapped onto such a machine.

For the other tasks ($2 \leq i \leq n + 1$ and $n + 3 \leq i \leq 2n + 2$), we have:

- $w_i = 1/C^2$ (i.e., the period is always matched);
- $f_{i,2n+1} = f_{i,2n+2} = 0$ (i.e., no failure on M_{2n+1} and M_{2n+2});
- $f_{i,u} = 1 - \frac{1}{a_u^2}$ for $2 \leq i \leq n + 1$ and $1 \leq u \leq 2n$;
- $f_{i,u} = 1 - \frac{1}{a_u}$ for $n + 3 \leq i \leq 2n + 2$ and $1 \leq u \leq 2n$.

$$\begin{array}{ccccccccccc}
\boxed{\frac{B}{C^2}} & \rightarrow & \multicolumn{3}{c}{\underbrace{\frac{1}{C^2} \rightarrow \cdots \rightarrow \frac{1}{C^2}}} & \rightarrow & \boxed{\frac{1}{B}} & \rightarrow & \multicolumn{3}{c}{\underbrace{\frac{1}{C^2} \rightarrow \cdots \rightarrow \frac{1}{C^2}}} \\
T_1 & & T_2 & \dots & T_{n+1} & & T_{n+2} & & T_{n+3} & \dots & T_{2n+2}
\end{array}$$

Clearly, the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$. We now show that $\mathcal{I}_1$ has a solution if and only if $\mathcal{I}_2$ does.

Suppose first that $\mathcal{I}_1$ has a solution, I. We build the following allocation for $\mathcal{I}_2$:

- T_1 is mapped on M_{2n+1};
- T_2 is mapped on M_{2n+2};
- for $n + 3 \leq i \leq 2n + 2$, T_i is mapped on a machine M_u, with $u \in I$;
- for $2 \leq i \leq n + 1$, T_i is mapped on a machine M_u, with $u \notin I$.

Note first that because of the values of $f_{i,u}$, the number of jobs to be computed for a task never exceeds C^2. Indeed, M_{2n+1} and M_{2n+2} never fail, and each task is mapped onto a distinct machine M_u, with $1 \leq u \leq 2n$, with a failure probability $f_u = 1 - \frac{1}{a_u^2}$ or $f'_u = 1 - \frac{1}{a_u}$. Note that $f'_u \leq f_u$ (indeed, $a_u \geq 1$ and $a_u^2 \geq a_u$), and therefore, for a task T_i, the number of products to compute is $x_i \leq \prod_{1 \leq u \leq 2n} 1/(1 - f_u) = \prod_{1 \leq u \leq 2n} a_u^2 = C^2$, see Section 2.3. Therefore, the period of machines $M_1, \dots, M_{2n}$, which are processing a task T_i with $w_i = 1/C^2$, is not greater than $x_i \times w_i = C^2 \times 1/C^2 = 1 = P$.

Now, we need to check the period of tasks T_1 and T_{n+2}. For T_{n+2}, we have $x_{n+2} = \prod_{n+2 \leq i \leq 2n+2} 1/(1 - f_{i,alloc(i)})$, where $M_{alloc(i)}$ is the machine on which T_i is mapped. Therefore, $x_{n+2} = \prod_{u \in I} a_u = B$, since I is a solution to $\mathcal{I}_1$. Since $w_{n+2,2n+2} = 1/B$, the period of machine M_{2n+2} is $B \times 1/B = 1 = P$. Finally, for T_1, $x_1 = \prod_{1 \leq i \leq 2n+2} 1/(1 - f_{i,alloc(i)}) = \prod_{u \notin I} a_u^2 \times \prod_{u \in I} a_u$. We have $\prod_{u \notin I} a_u^2 = (C/B)^2$, and therefore $x_1 = C^2/B$, and the period of machine M_{2n+1} is exactly 1 as well. We have a solution to $\mathcal{I}_2$.

Suppose now that $\mathcal{I}_2$ has a solution. It has to be a one-to-one mapping, since there are no more machines than tasks. If T_1 is not mapped on M_{2n+1}, or if T_{n+2} is not mapped on M_{2n+2}, the period of the corresponding machine is at least $2P$, and hence the solution is not valid. The other tasks are therefore mapped on machines $M_1, \dots, M_{2n}$. Let I be the set of n machines on which tasks $T_{n+3}, \dots, T_{2n+2}$ are mapped. The period for task T_{n+2} is respected, and therefore, $\prod_{u \in I} a_u \times \frac{1}{B} \leq 1$, and

$$\prod_{u \in I} a_u \leq B \; .$$

Then, for the period of task T_1, we obtain $\prod_{u \notin I} a_u^2 \times \prod_{u \in I} a_u \times \frac{B}{C^2} \leq 1$, therefore $\prod_{u \notin I} a_u \times C \times \frac{B}{C^2} \leq 1$, and finally

$$\prod_{u \notin I} a_u \leq \frac{C}{B} \; .$$

Since $\prod_{u \in I} a_u \times \prod_{u \notin I} a_u = C$, the two inequalities above should be tight, and therefore $\prod_{u \in I} a_u = B$, which means that $\mathcal{I}_1$ has a solution. This concludes the proof. $\square$

3.4 Fixed allocation of tasks to machines

If the allocation of tasks to machines is known, then we can optimally decide how to share tasks between machines, in polynomial time. We build upon the linear program of Theorem 4, and we add a set of parameters: $a_{i,u} = 1$ if T_i is allocated to M_u, and $a_{i,u} = 0$ otherwise (for $1 \leq i \leq n$ and $1 \leq u \leq m$). The variables are still the period P, and the amount of task per machine $q(i,u)$. The linear program writes:

$$\begin{array}{lrl}
\text{Minimize } P & & \\
\text{subject to} & \text{(i)} & q(i,u) \geq 0 \text{ for } 1 \leq i \leq n, 1 \leq u \leq m \\
& \text{(iia)} & \displaystyle\sum_{1 \leq u \leq m} q(n,u) \times (1 - f_{n,u}) = 1 \\
& \text{(iib)} & \displaystyle\sum_{1 \leq u \leq m} q(i,u) \times (1 - f_{i,u}) = \sum_{1 \leq u \leq m} q(i+1,u) \text{ for } 1 \leq i < n \\
& \text{(iii)} & \displaystyle\sum_{1 \leq i \leq n} q(i,u) \times w_{i,u} \leq P \text{ for } 1 \leq u \leq m \\
& \text{(iv)} & q(i,u) \leq a_{i,u} \times F_{\max} \text{ for } 1 \leq i \leq n \text{ and } 1 \leq u \leq m
\end{array} \tag{3}$$

We have added constraint (iv), which states that $q(i,u) = 0$ if $a_{i,u} = 0$, i.e., it enforces that the fixed allocation is respected. $F_{\max} = \prod_{1 \leq i \leq n} \max_{1 \leq u \leq m} f_{i,u}$ is an upper bound on the $q(i,u)$ values, it can be pre-computed before running the linear program. The size of this linear program is clearly polynomial in the size of the instance, all $n \times m + 1$ variables are rational, and therefore it can be solved in polynomial time [11].

3.5 Integer linear program

The linear program of Equation (3) allows us to find the solution in polynomial time, once the allocation is fixed. We also propose an integer linear program (ILP), which computes the solution to the MinPer($spe, f_{i,u}, w_{i,u}$) problem, even if the allocation is not known. However, because of the integer variables, the resolution of this program takes an exponential time. Note that this ILP can also solve the MinPer($o2m, f_{i,u}, w_{i,u}$): one just needs to assign a different type to each task.

Compared to the linear program of the previous section, we no longer have the $a_{i,u}$ parameters, and therefore we suppress constraint (iv). Rather, we introduce a set of Boolean variables, $x(u,t)$, for $1 \leq u \leq m$ and $1 \leq t \leq p$, which is set to 1 if machine M_u is specialized in type t, and 0 otherwise. We then add the following constraints:

(iva) $\sum_{1 \leq t \leq p} x(u,t) \leq 1$ for each machine M_u with $1 \leq u < m$;

(ivb) $q(i,u) \leq x(u,t_i) \times F_{\max}$ for $1 \leq i \leq n$ and $1 \leq u \leq m$.

Constraint (iva) states that each machine is specialized into at most one type, while constraint (ivb) enforces that $q(i,u) = 0$ when machine M_u is not specialized in the type t_i of task T_i.

This ILP has $n \times m + 1$ rational variables, and $m \times p$ integer variables. The number of constraints is polynomial in the size of the instance. Note that this ILP can be solved for small problem instances with ILOG CPLEX [4].

4 Heuristics and simulations

From the complexity study of Section 3, we are able to find an optimal mapping for $\textsc{MinPer}(gen, *, *)$. In this section, we provide practical solutions to solve $\textsc{MinPer}(spe, f_{i,u}, w_{i,u})$, which is NP-hard. Indeed, general mappings are not feasible in some cases, since it involves reconfiguring the machines between the execution of two tasks whose type is different. This additional setup time may be unaffordable. We design in Section 4.1 a set of polynomial time heuristics which return a specialized mapping, building upon the complexity results of Section 3. Finally, we present exhaustive simulation results in Section 4.2.

4.1 Polynomial time heuristics

Since we are able to find the optimal solution once the tasks are mapped onto machines, the heuristics are building such an assignment, and then we run the linear program of Section 3.4 to obtain the optimal solution in terms of $q(i, u)$. The first heuristic is random, and serves as a basis for comparison. Then, the next three heuristics (H2, H3 and H4) are based on an iterative allocation process in two stages. In the first *top-down* stage, the machines are assigned from task T_1 to task T_n depending on their speed $w_{i,u}$: the machine with the best $w_{1,u}$ is assigned to T_1 and so on. The motivation is that the workload of the first task is larger than the last task because of the job failures that arise along the pipeline. In the second *bottom-up* stage, the remaining machines are assigned from task T_n to task T_1 depending on their reliability $f_{i,u}$: the machine with the best $f_{n,u}$ is assigned to T_n and so on. The motivation is that it is more costly to lose a job at the end of the pipeline than at the beginning, since more execution time has been devoted to it. We iterate until all the machines have at least one task to perform. Finally, H5 performs only a *top-down* stage, repetitively. The heuristics are described below.

H1: Random heuristic. The first heuristic randomly assigns each task to a machine when the allocation respects the task type of the chosen machine.

H2: without any penalization. The *top-down* stage assigns each task to the fastest possible machine. At the end of this stage, each task of the same type is assigned onto the same machine, the fastest. Then, the already assigned machines are discarded from the list. In the same way, the *bottom-up* stage assigns each task of the same type to the same machine starting from the more reliable one. We iterate on these two steps until all machines are specialized.

H3: workload penalization. The difference with H2 is in the execution of the *top-down* stage. Each time a machine is assigned to a task, this machine is penalized to take the execution of this task into account and its $w_{i,u}$ is changed to $w_{i,u} \times (k+1)$ where k is the number of tasks already mapped on the machine M_u. This implies that several machines can be assigned to the same task type in this phase of the algorithm: if a machine is already loaded by several tasks then we may find a faster machine and assign it to this task type. The *bottom-up* stage has the same behavior as for H2.

H4: cooperation work. In this heuristic, a new machine is assigned to each task, depending on its speed, during the *top-down* stage then the *bottom-up* stage has the same behavior as the heuristic H2.

H5: Focus on speed. The heuristic H5 focuses only on the speed by repeating the *top-down* stage previously presented in the heuristic H3 until all the machines are allocated to at least one task.

4.2 Simulations

In this section, we evaluate the performance of the five heuristics. The period returned by each heuristic is measured in *ms*. Recall that m is the number of machines, p the number of types, and n the number of tasks. Each point in a figure is an average value of 30 simulations where the $w_{i,u}$ are randomly chosen between 100 and 1000 *ms*, for $1 \leq i \leq n$ and $1 \leq u \leq m$, unless stated otherwise. Similarly, failure rates $f_{i,u}$ are randomly chosen between 0.2 and 10 % (i.e., 1/500 and 1/10), unless stated otherwise.

Heuristics versus linear program. In this set of experiments, the heuristics are compared to the integer linear program which gives the optimal solution. The platform is such that $m = 20$, $p = 5$ and $21 \leq n \leq 61$. Figure 5 shows that the random heuristic H1 has poor performance. Therefore, for visibility reasons, H1 does not appear in the rest of the figures.

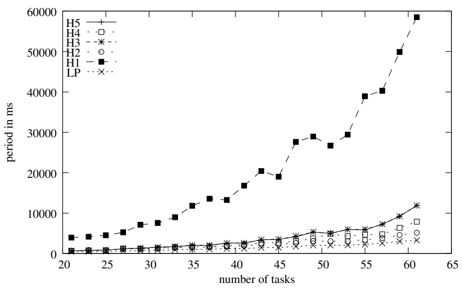

Figure 5: $m = 20$, $p = 5$.
Heuristics against the linear program.

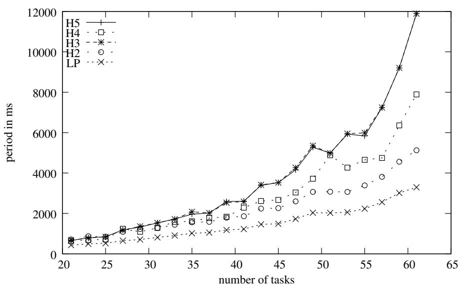

Figure 6: $m = 20$, $p = 5$.
Without H1.

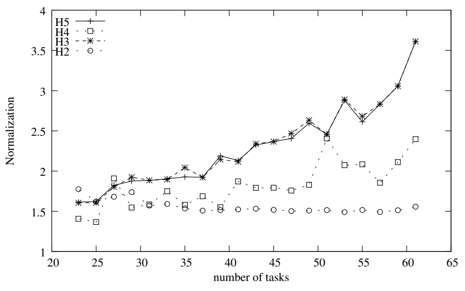

Figure 7: $m = 20$, $p = 5$.
Heuristics against the linear program - Normalization.

We focus in Figure 6 on the heuristics H2 to H5. Results show that the heuristics are not far from the optimal. The same experiment is used in Figure 7 with the normalization of the heuristics upon the linear program. The best heuristics H2 and H4 are between 1.5 and 2 from the optimal solution. With this configuration, although exponential, the linear program always finds a result. Note that, for a platform with 20 machines and 10 types instead of 5, the percentage of success of the linear program is less than 50% with 61 tasks.

General behavior of the heuristics. In a second set of simulations, we focus on the behavior of the heuristics alone. First we focus on the difference between having more tasks than machines or the contrary. A platform with 50 machines and 25 types of tasks is set. In Figure 8, the number of tasks varies between 10 and 50. Results show that H2 is slightly less performing than the other heuristics. This difference increases as the number of types get closer to the number of machines, as shown in Figure 9.

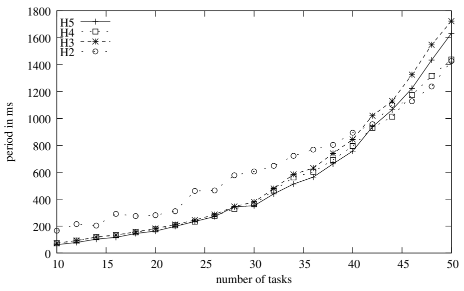

Figure 8: $m = 50$, $p = 25$.
Heuristics with more machines than tasks.

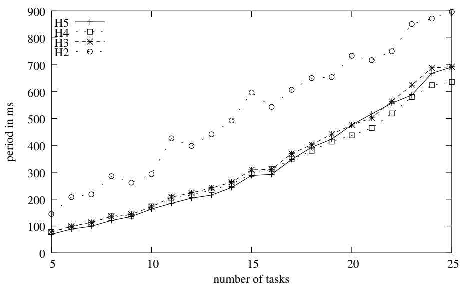

Figure 9: $m = 25$, $p = 15$.
Heuristics with more machines than tasks.

When the number of tasks is higher than the number of machines, H2 and H4 become clearly the best heuristics (see Figure 10). Indeed, at the end of the first allocation stage, H3 and H5 will almost have used all the machines and the second stage will thus not be decisive. This is why the lines of H3 and H5 are superimposed in this case.

To study the impact of the speed of the machines, we set a platform with almost homogeneous machines ($100 \leq w_{i,u} \leq 200$). Results are presented in Figure 11 and shows that the homogeneity in terms of the machine speeds does not change the overall comportment of the heuristics. H2 and H4 are still the best even if the gap with H3 and H5 is reduced.

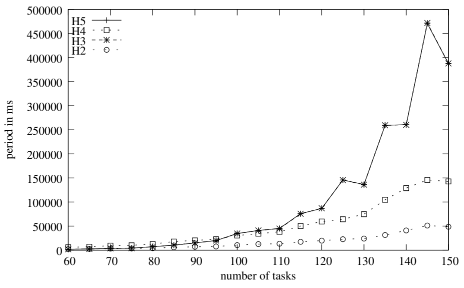

Figure 10: $m = 50$, $p = 25$.
Heuristics with more tasks than machines.

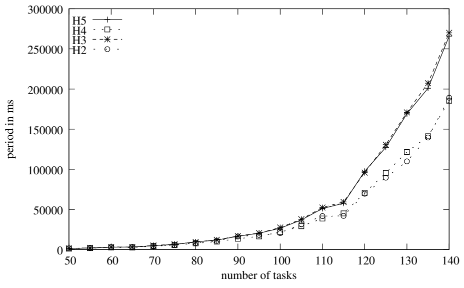

Figure 11: $m = 40$, $p = 30$, $100 \leq w_{i,u} \leq 200$.
Homogeneous machines.

To conclude with general behavior, we studied the impact of the number of types with two platforms of 40 machines and a number of tasks ranging from 10 to 110. The number of task types is set to 5 for the first one (Figure 12) and to 35 for the second one (Figure 13). When the number of types is small compared to the number of machines (Figure 12), the opportunity to split groups is high. In this case, H2 and H4 are the best heuristics because the workload is shared among a bigger set of machines and not only on those that are efficient for a given task. In the contrary, when the number of types is close to the number of machines, the number of split tasks decrease. Indeed, each machine must be specialized to one type. In the experiment shown in Figure 13, only 5 machines can be used to share the workload once each machine is dedicated to a type. That explains why the performances of the heuristics are pretty much alike.

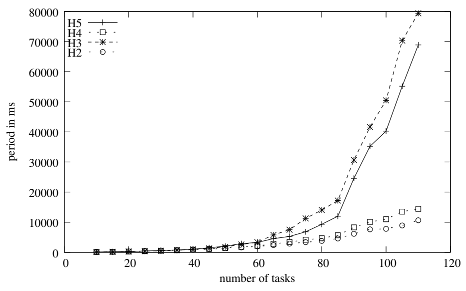

Figure 12: $m = 40$, $p = 5$.
Small number of types.

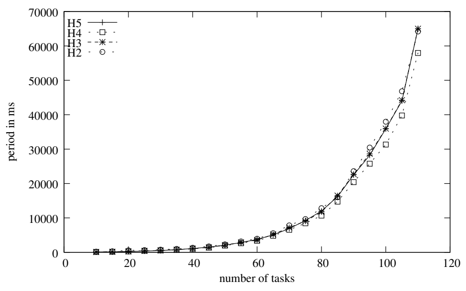

Figure 13: $m = 40$, $p = 35$.
High number of types.

Impact of the failure rate. In this last set of simulations, we study the impact of the failure rate on the heuristics. Figures 14 and 15 show that when the failure rate is high ($0 \leq f_{i,u} \leq 30\%$), only H2 and H4 have a good performance. Remember that our heuristics have two stages, the first one optimizes the $w_{i,u}$ and the second one the $f_{i,u}$. In the case of H3 and H5, the first stage does not encourage the reuse of a machine already assigned to a task (the penalization is high). Thus, in the particular case of the platform set in Figures 14 and 15, H3 and H5 assign all the machines at the end of their first stage and cannot optimize the failure in the second stage because no more machines are available.

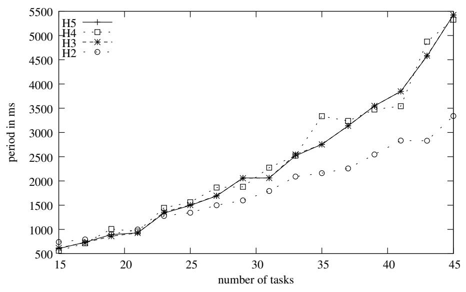

Figure 14: $m = 15$, $p = 5$.
Failure $0 \leq f_{i,u} \leq 10\%$.

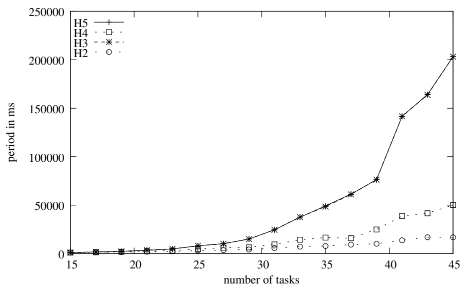

Figure 15: $m = 15$, $p = 5$.
Failure $0 \leq f_{i,u} \leq 30\%$.

Summary. Even though it is clear that H1 performs really poorly, the other heuristics can all be the most appropriate, depending upon the situation. Note that the comparison between the heuristics is made easier if the gap between the number of types and the number of machines is big. Indeed, with a small number of types, the tasks can be split many times because more machines are potentially dedicated to a same type. The choices made by a heuristic either to split a task or not have more impact on the result.

5 Conclusion

In this paper, we investigate the problem of maximizing the throughput of coarse-grain pipeline applications where tasks have a type and are subject to failures. We propose mapping strategies to solve the problem considering three rules of the game: one-to-many mappings (each machine processes at most one task), specialized mappings (several tasks of the same type per machine), or general mappings. In any case, the jobs associated to a task can be distributed upon the platform so as to balance workload between the machines. From a theoretical point of view, an exhaustive complexity study is proposed. We prove that an optimal solution can be computed in polynomial time in the case of general mappings whatever the application/platform parameters, and in the case of one-to-many and specialized mappings when the faults only depend on the tasks, while our optimization problem becomes NP-hard in any other cases. Since general mappings do not provide a realistic solution because of unaffordable setup times when reconfiguration occurs, we propose to solve the specialized mapping problem by designing several polynomial heuristics. Also, we give an integer linear programming formulation of the problem that allows us to compute an optimal solution on small problem instances and to evaluate the performance of these heuristics on such instances. The simulations show that some heuristics return specialized mappings with a throughput close to the optimal, and that using random mappings never gives good solutions. As future work, we plan to investigate other objective functions, as the latency, or other failure models in which the failure rate associated to the task and/or the machine is correlated with the time to perform that task.

Acknowledgment

Anne Benoit is with Institut Universitaire de France. This work was supported in part by the ANR StochaGrid and RESCUE projects.

References

[1] J. Bahi, S. Contassot-Vivier, and R. Couturier. Coupling dynamic load balancing with asynchronism in iterative algorithms on the computational grid. In *International Parallel and Distributed Processing Symposium, IPDPS 2003*, Apr. 2003.

[2] J. Błażewicz, M. Drabowski, and J. Weglarz. Scheduling multiprocessor tasks to minimize schedule length. *IEEE Trans. Comput.*, 35:389–393, May 1986.

[3] W. Cirne, F. Brasileiro, D. Paranhos, L. F. W. Góes, and W. Voorsluys. On the efficacy, efficiency and emergent behavior of task replication in large distributed systems. *Parallel Computing*, 33(3):213–234, 2007.

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

[5] E. Descourvières, S. Debricon, D. Gendreau, P. Lutz, L. Philippe, and F. Bouquet. Towards automatic control for microfactories. In *IAIA'2007, 5th Int. Conf. on Industrial Automation.*, 2007.

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

[7] H. Gröflin, A. Klinkert, and N. P. Dinh. Feasible job insertions in the multi-processor-task job shop. *European J. of Operational Research*, 185(3):1308 – 1318, 2008.

[8] P. Jalote. *Fault Tolerance in Distributed Systems.* Prentice Hall, 1994.

[9] A. Litke, D. Skoutas, K. Tserpes, and T. Varvarigou. Efficient task replication and management for adaptive fault tolerance in mobile grid environments. *Future Generation Computer Systems*, 23(2):163 – 178, 2007.

[10] B. Parhami. Voting algorithms. *IEEE Trans. on Reliability*, 43(4):617–629, 1994.

[11] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*, volume 24 of *Algorithms and Combinatorics.* Springer-Verlag, 2003.

[12] M. Tanaka. Development of desktop machining microfactory. *Journal RIKEN Rev*, 34:46–49, April 2001. ISSN:0919-3405.

[13] I. Verettas, R. Clavel, and A. Codourey. Pocketfactory: a modular and miniature assembly chain including a clean environment. In *5th Int. Workshop on Microfactories*, 2006.

[14] J. B. Weissman and D. Womack. Fault tolerant scheduling in distributed networks, 1996.

[15] R. West and C. Poellabauer. Analysis of a window-constrained scheduler for real-time and best-effort packet streams. In *Proc. of the 21st IEEE Real-Time Systems Symp.*, pages 239–248. IEEE, 2000.

[16] R. West and K. Schwan. Dynamic Window-Constrained Scheduling for Multimedia Applications. In *ICMCS, Vol. 2*, pages 87–91, 1999.

[17] R. West, Y. Zhang, K. Schwan, and C. Poellabauer. Dynamic window-constrained scheduling of real-time streams in media servers, 2004.

[18] M. Wieczorek, A. Hoheisel, and R. Prodan. Towards a general model of the multi-criteria workflow scheduling on the grid. *Future Gener. Comput. Syst.*, 25(3):237–256, 2009.

[19] J. Yu and R. Buyya. A taxonomy of workflow management systems for grid computing. Research Report GRIDS-TR-2005-1, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia, Apr 2005.

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