Colocating tasks in data centers using a side-effects performance model

Fanny Pascual[a], Krzysztof Rzadca[b]

[a]*Sorbonne Universités, UPMC (Université Paris 6), LIP6, CNRS, UMR 7606, F-75005, Paris, France*
[b]*Institute of Informatics, University of Warsaw Warsaw, Poland*

Abstract

In data centers, many tasks (services, virtual machines or computational jobs) share a single physical machine. We explore a new resource management model for such colocation. Our model uses two parameters of a task—its size and its type—to characterize how a task influences the performance of the other tasks allocated on the same machine. As typically a data center hosts many similar, recurring tasks (e.g. a webserver, a database, a CPU-intensive computation), the resource manager should be able to construct these types and their performance interactions. In particular, we minimize the total cost in a model in which each task's cost is a function of the total sizes of tasks allocated on the same machine (each type is counted separately). We show that for a linear cost function the problem is strongly NP-hard, but polynomially-solvable in some particular cases. We propose an algorithm polynomial in the number of tasks (but exponential in the number of types and machines) and another algorithm polynomial in the number of tasks and machines (but exponential in the number of types and admissible sizes of tasks). We also propose a polynomial time approximation algorithm, and, in the case of a single type, a polynomial time exact algorithm. For convex costs, we prove that, even for a single type, the problem becomes NP-hard, and we propose an approximation algorithm. We experimentally verify our algorithms on instances derived from a real-world data center trace. While the exact algorithms are infeasible for large instances, the approximations and heuristics deliver reasonable performance.

Keywords: Scheduling, Combinatorial optimization, Data center, Heterogeneity, Colocation

1. Introduction

Data centers, composed of tens to hundreds of thousands of machines, packaged as virtual machines or services and sold under the label of cloud, are now changing the way the industry (and, to some extent, academia and research) computes. Virtualization packages individual resources into standard chunks with performance guaranteed by Service Level Agreements (SLAs). Economies of scale make the whole endeavor profitable for huge companies, like Google, or providers of for-hire computational power (such as Amazon EC2, RackSpace or Google Compute Engine).

Email addresses: fanny.pascual@lip6.fr (Fanny Pascual), krz@mimuw.edu.pl (Krzysztof Rzadca)

There are significant differences between a data center and standard High Performance Computing (HPC) machines. In their great majority, HPC workloads are composed of computationally-intensive batch jobs only (although some recent HPC workloads may also be memory-intensive, which requires changes to HPC resource managers [Klusácek and Rudová, 2014]). The goal of an HPC scheduler is to order jobs so that they are completed as fast as possible, taking into account site's policies, fairness and efficiency. As jobs are computationally-intensive, they all compete for the same resource—the CPU. So, a single node executes at most as many jobs as CPU cores.

In contrast, a data center workload is more varied. In the Google trace [Reiss et al., 2012], just 1.5% of applications contribute 98.5% of CPU usage [Di et al., 2014]. Thus, while there are some computationally-intensive batch jobs (corresponding to, e.g., Pagerank recalculation), a large part of the workload is services. Services are varied, from user-facing web applications to databases to message-passing infrastructure. We will use the term *task* to denote a single instance of a service or a single job. We assume that each task executes on a single physical machine.

These new features of data center workloads make HPC models unsuitable for managing resources of a data center. As tasks require heterogeneous resources [Reiss et al., 2012] (CPU, memory, hard disk bandwidth, network bandwidth), sharing a single machine among many services is reasonable. Ideally, resource requirements of colocated tasks should complement each other, e.g., a memory-intensive database instance should be allocated with a few IO-intensive web applications with burst popularity. The goal of the resource manager is also different: instead of completing tasks as fast as possible, the resource manager should optimize the end-user experience (e.g., a statistic of the response time such as time by which 95 or 99 percent of requests are completed).

The objective of this work is to explore an alternative model of data center resource management that captures both complex goal functions and the complex performance relations that tasks have on each other when they are allocated to the same machine. In classic scheduling, a task's influence on other tasks depends solely on its size (which represents its load, or its processing time). We propose a notion of a *type* of a task. Each task influences the performance of other tasks allocated to the same machine. We call such influence the side-effects. The influence is a function of the size *and the type* of a task. Thus, the performance of a task assigned to a machine M is a function of the size of the tasks of each type on that machine. If there are T possible types, this function takes T arguments. The i-th argument is the total size of the tasks of type i on M.

The paper is organized as follows. We define the problem of Partition with Side Effects (PSE) in Section 2. Sections 3–7 contain our theoretical results and Section 8 experimental results. In Section 3 we show that PSE is NP-complete. In Section 4 we present a dominance rule (for any instance, there is an optimal schedule in which the tasks are ordered by size). In Section 5 we give an optimal polynomial time algorithm for PSE with a single type. For the general PSE, we propose two optimal algorithms (polynomial in some parameters, exponential in some others), and one approximate algorithm in Section 6. In Section 7 we analyze a variant of PSE with strictly convex cost functions (a function $f : X \to \mathcal{R}$ is strictly convex if for all $x \neq y \in X$, for all $t \in (0, 1)$, $f(tx + (1 - t)y) < tf(x) + (1 - t)f(y)$). We study the performance of the proposed algorithms by simulation in Section 8. The paper concludes with a brief discussion of related work (Section 9) and a summary of our results (Section 10).

2. Theory: proposed colocation model

We consider a system where n tasks $J = \{1, \ldots, n\}$ have to be allocated on a set of m parallel identical machines $\mathcal{M} = \{M_1, \ldots, M_m\}$. Each task i has a known size $p_i \in \mathbb{N}$. This assumption is analogous to the clairvoyance assumption in the standard scheduling theory. The sizes can be estimated by the resource manager using previous instances, or users' estimates. The size corresponds to the load the task imposes on a machine: for instance, the request rate for a web server or the CPU load for a CPU-intensive computation. We assume that the tasks are indexed by non-increasing sizes: $p_1 \geq p_2 \geq \cdots \geq p_n$.

A *partition* (also called an *allocation*) is an assignment of each of the n tasks to one of the m machines. In other words, a partition divides the tasks into at most m subsets, each subset corresponding to the tasks allocated on the same machine. Given a partition P, we denote by $M_{P,i} \in \mathcal{M}$ the machine on which task i is allocated in P.

The main contribution of this paper lies in analyzing side-effects of colocating tasks on a single machine. Each task influences the performance of other tasks allocated to the same machine. Specifically, the impact of task i on the performance of another task j is a function of the task's size p_i and the *task's type* t_i. Types generalize tasks' impact on the performance. The operator of the data center should define types according to observed performance dependencies. A type could correspond to a specific application (as in [Kim et al., 2015]); but it could also be more general, gathering, for instance, all webservers under a single type, and all databases under another one. Let $\mathcal{T} = \{1, \ldots, T\}$ be a set of T different types of tasks. Each task i has type $t_i \in \mathcal{T}$. Here we assume that the task's type is known to the resource manager either by analysis of previous instances, or by users' declarations. For each type $t \in \mathcal{T}$, we denote by J^t the tasks which are of type t; by $n^{(t)}$ the number of such tasks ($n^{(t)} = |J^t|$); by j_i^t the i-th largest task of type t (ties are broken arbitrarily); and by p_i^t this task's size.

Different types have different influence on the performance of a task. We model performance by cost (unrelated to the monetary cost). In most of the paper (except Section 7) we use a linear cost function, i.e., the cost c_i of task i varies linearly with the total load of tasks j colocated on the same machine $M_{P,i}$. Different types have different impacts. More precisely,

$$c_i = \sum_{j \text{ on machine } M_{P,i}} p_j \alpha_{t_j,t_i}, \tag{1}$$

where a coefficient $\alpha_{t,t'} \in \mathbb{N}$, defined for each pair of types $(t, t') \in \mathcal{T}^2$, measures the impact of the tasks of type t on the cost of the tasks of type t' (allocated on the same machine). If $\alpha_{t,t'} = 0$ then a task of type t has no impact on the cost of a task of type t'. The higher the $\alpha_{t,t'}$, the larger the impact. Coefficients are not symmetric, i.e., it is possible that $\alpha_{t,t'} \neq \alpha_{t',t}$. We consider the linear cost function as it generalizes, by adding coefficients $\alpha_{t,t'}$, one of well-known scheduling models [Koutsoupias and Papadimitriou, 1999, Vöcking, 2007], in which the cost of a task is the load of its machine (i.e., if $\forall (t, t') \in \mathcal{T}^2$ $\alpha_{t,t'} = 1$, the model reduces to the classic model). The coefficients $\alpha_{t,t'}$ can be estimated by monitoring tasks' performance as a function of their colocation and their sizes, which should be feasible as a data center runs many instances of similar services [Kim et al., 2015, Podzimek et al., 2015].

We denote by PSE (Partition with Side Effects) the problem of finding a partition P^* minimizing the total cost $C(P) = \sum_{i=1}^{n} c_i$, with c_i defined by the linear cost function. The partition with the minimal cost minimizes the average cost of a task and thus corresponds to the socially-optimal outcome in the utilitarian model.

In Section 7, we study an important generalization of the cost model, in which the cost of a task is any convex function of the total load of the machine.

3. Complexity

The notion of type increases the complexity of the allocation problem. When all tasks have the same type, our problem reduces to the problem of [Koutsoupias and Papadimitriou, 1999, Vöcking, 2007]. As we show in Section 5, the single type problem can be solved in polynomial time. In contrast, in this section we show that with multiple types PSE is strongly NP-complete even for two machines and unit-size tasks. The following result is a strong computational motivation for the introduction of the notion of a type. If the problem was polynomial with a large number of types, we could have a type for each job. However, this result shows that it is beneficial from the computational perspective to generalize a task's influences into a small number of distinct types.

Proposition 3.1. *The decision version of* PSE *is strongly NP-complete, and this even if there are only two machines and if the tasks have unit size. Moreover, there is no polynomial time r-approximate algorithm for* PSE*, for any constant* $r > 1$*, unless* $P = NP$.

Proof. We reduce the strongly NP-hard Simple Min UnCut problem to PSE. The Simple Min UnCut problem [Garg et al., 1993] is the following one: given a graph $G = (V, E)$ and a positive integer K, is there a partition of V into two disjoints sets V_1 and V_2 such that the number of edges whose endpoints are both in the same set is at most K? Note that this problem is a complement to the well-known NP-complete Simple Max Cut problem [Garey and Johnson, 1979]: the sum of the number of cut edges and uncut edges is equal to the total number of edges in the graph. Therefore, the Simple Min UnCut problem is a strongly NP-hard problem.

The decision version of PSE is as follows: given an instance of PSE, and a bound (positive integer) B, is there a partition with cost at most B? We construct an instance of PSE from an instance of Simple Min UnCut as follows: we have two machines M_1 and M_2, and $n = |V|$ tasks $\{1, \ldots, n\}$, each one of size 1. We label the vertices of V by the integers from 1 to n. Each task i corresponds to vertex i of V. There are n types (one per task). For each $i \in \{1, \ldots, n\}$, we set $t_i = i$. The values α are defined as follows: for each $(i, j) \in V^2$, $\alpha_{i,j} = \frac{1}{2}$ if $\{i, j\} \in E$ and $\alpha_{i,j} = 0$ if $\{i, j\} \notin E$. We fix $B = K$.

Let us show that there is a solution of cost K of this instance of PSE if and only if there is a solution of cost K for the corresponding instance of Simple Min UnCut.

Assume first that there is a solution of cost K to Simple Min UnCut: let V_1 and V_2 be two sets such that the number of edges whose both endpoints are in the same set is K. We construct a partition P for PSE by assigning the tasks corresponding to vertices in V_1 to machine M_1, and the tasks of V_2 to machine M_2.

Let $i \in \{1, \ldots, n\}$. For each task i, let $n_i = \sum_{j \text{ on } M_{P,i} | \{i,j\} \in E} 1$. In other words, for each task i on M_1, n_i is the number of neighbors of vertex i in V_1, and for each task i on M_2, n_i is the number of neighbors of vertex i in V_2. The cost of task i is $c_i = \sum_{j \text{ on } M_{P,i}} \alpha_{j,i} p_j = \frac{1}{2} \sum_{j \text{ on } M_{P,i} | \{i,j\} \in E} 1 = \frac{1}{2} n_i$. The total cost of partition P is $C(P) = \sum_{i=1}^{n} c_i = \frac{1}{2} \sum_{i=1}^{n} n_i$. Note that $\sum_{i=1}^{n} n_i$ is twice the number of edges for which both endpoints are in the same set (an edge $\{i, j\}$ between two vertices in the same set adds 1 to n_i and 1 to n_j). Since the number of edges for which both endpoints are in the same set is K, we have $C(P) = K$. There is thus a solution of cost K to PSE.

Assume now that there is a partition P of cost K for PSE. We construct a partition P of V by defining $V_j = \{i \in V : \text{task } i \text{ is assigned to } M_j\}$. The total cost $C(P)$ is equal to the sum of the costs of the tasks. The cost of a task i in P is $\sum_{j \text{ on } M_{P,i}} \alpha_{j,i} p_j = \frac{1}{2} \sum_{j \text{ on } M_{P,i} | \{i,j\} \in E} 1$. Therefore, the cost of P is equal to the number of edges which have both endpoints either in V_1 or in V_2, that is K: there is a solution of cost K to Simple Min UnCut.

We have shown that there is a solution of cost K to Simple Min UnCut if and only if there is a solution of cost K to PSE. As PSE is (trivially) in NP, and as Simple Min UnCut is strongly NP-complete, the decision version of PSE is also strongly NP-complete. Moreover, it has been shown that the Min UnCut[1] problem, which is the optimization version of Simple Min UnCut is APX-hard [Creignou et al., 2001]. An APX-hard problem cannot be approximated within a constant factor by a polynomial time algorithm, unless $P = NP$. Above, we show that the cost of an optimal solution of an instance of Min UnCut has the same cost that an optimal solution of the corresponding instance of PSE. Thus, PSE is also hard to approximate within some constant factor. □

The best approximate algorithm for the Min UnCut problem is $O(\sqrt{\log\ n})$-approximate [Agarwal et al., 2005]. Note that the reduction used in the proof above shows that a polynomial time r-approximate algorithm for PSE would imply a polynomial time r-approximate algorithm for the Min UnCut problem.

4. The ordered sizes (OrS) property

We propose a dominance rule called *the* OrS *(Ordered Sizes) property*. Take three tasks s (small), x (medium), l (large) of the same type and of sizes $p_s < p_x < p_l$. An allocation breaks the OrS property if s and l are assigned to the same machine and x to another machine. An allocation fulfills the OrS property if no triple breaks it. We show that there is an optimal allocation that fulfills this OrS property.

We will use this result in a dynamic programming algorithm, BestOrS, which is optimal when there is a single type (see Section 5), and in an algorithm CutJuxtapose (Section 6.3), which is polynomial in the number of tasks, but exponential in the number of types and machines.

Lemma 4.1. *For each instance of* PSE *there exists an optimal allocation which fulfills the* OrS *property.*

Proof. The proof is by contradiction. Assume that there is an instance I for which there is no optimal allocation which fulfills the OrS property. Let P be an optimal allocation for I such that there is a minimal number of triples which break the OrS property. As P does not fulfill the OrS property, there are three tasks s (small) x (medium) and l (large) of type $t \in \mathcal{T}$ such that: (1) $p_s < p_x < p_l$; and (2) the tasks s and l are in P on the same machine, M_i, and the task x is on another machine $M_j \neq M_i$. Let us denote by P_{s-x} an allocation in which s and x are exchanged (i.e., task s is on M_j, task x is on M_i, and the remaining tasks $k \notin \{s, x\}$ are on the same machines as in P, $M_{P_{s-x},k} = M_{P,k}$). Likewise, we denote by P_{l-x} an allocation in which l and x are exchanged. We will now show that the costs of the partitions P_{s-x} and P_{l-x} are equal to the cost of P. This result will lead to a contradiction, as either in P_{s-x} or in P_{l-x} the number of

[1]The Min UnCut problem is the following one: given a graph $G = (V, E)$, find a partition of V into two disjoints sets V_1 and V_2 such that the number of edges whose both endpoints are in the same set is minimized.

triples which break the OrS property is strictly smaller than the number of triples which break the OrS property in P.

For each type $q \in \mathcal{T}$, and for each machine $M \in \mathcal{M}$, the number of tasks of type q on M is the same in P, P_{s-x}, and P_{l-x}. Let n_i^q and n_j^q denote the numbers of tasks of type q on M_i and M_j.

For each task a which is not allocated to M_i or M_j in P, the cost of a is the same in P, P_{s-x}, and P_{l-x}. Indeed, the cost of a task depends only on the tasks allocated on the same machine, and the partitions P, P_{s-x}, and P_{l-x} are identical on all the machines except machines M_i and M_j. In P_{s-x} on M_i, the loads of the other types remain the same, so the cost of each task of type q increases exactly by $(p_x - p_s)\alpha_{t,q}$ compared to its cost in P, since the exchange of s and x *increases* on M_i the load of type t by $(p_x - p_s)$. Likewise, in P_{s-x} on M_j, the cost of each task of type q decreases by $(p_x - p_s)\alpha_{t,q}$ compared to its cost in P, since the exchange of s and x *decreases* on M_j the load of type t by $(p_x - p_s)$. Therefore, we have:

$$\begin{aligned} C(P_{s-x}) &= C(P) + \sum_{q\in\mathcal{T}} \left((p_x - p_s)\alpha_{t,q} n_i^q\right) + \\ &\quad + \sum_{q\in\mathcal{T}} \left((p_s - p_x)\alpha_{t,q} n_j^q\right) \\ &= C(P) + (p_x - p_s) \sum_{q\in\mathcal{T}} \left(\alpha_{t,q}(n_i^q - n_j^q)\right). \end{aligned} \tag{2}$$

Likewise,

$$\begin{aligned} C(P_{l-x}) &= C(P) + \sum_{q\in\mathcal{T}} \left((p_x - p_l)\alpha_{t,q} n_i^q\right) + \\ &\quad + \sum_{q\in\mathcal{T}} \left((p_l - p_x)\alpha_{t,q} n_j^q\right) \\ &= C(P) + (p_x - p_l) \sum_{q\in\mathcal{T}} \left(\alpha_{t,q}(n_i^q - n_j^q)\right). \end{aligned} \tag{3}$$

If $\sum_{q\in\mathcal{T}} \left(\alpha_{t,q}(n_i^q - n_j^q)\right) < 0$ then $C(P_{s-x}) < C(P)$, which is impossible since P is optimal. Likewise, if $\sum_{q\in\mathcal{T}} \left(\alpha_{t,q}(n_i^q - n_j^q)\right) > 0$ then $C(P_{l-x}) < C(P)$, which again is impossible since P is optimal. Thus,

$$\sum_{q\in\mathcal{T}} \left(\alpha_{t,q}(n_i^q - n_j^q)\right) = 0, \tag{4}$$

and $C(P) = C(P_{l-x}) = C(P_{s-x})$.

□

5. Special case: single type

We consider in this section that all the tasks are of the same type, t_1. We show a polynomial time algorithm, based on dynamic programming, for allocating tasks to machines. This algorithm, called BestOrS, is optimal for a single type and linear costs. Thus, it finds the socially-optimal outcome in the [Koutsoupias and Papadimitriou, 1999] model in which the cost of each

task is the load of its machine. This algorithm will also allow us to solve the special cases of PSE detailed in the introduction of Section 6.

The cost of each task i is $c_i = L_{M_{P_i}}$, the load of the machine on which task i is allocated (where $L_j = \sum_{i \text{ on machine } j} p_i$). For a single type, the cost of a partition P would be $\alpha_{t_1,t_1} \sum_{i=1}^{n} L_{M_{P_i}}$. Therefore, without loss of generality, we fix $\alpha_{t_1,t_1} = 1$.

The following dynamic programming algorithm, BestOrS, finds in polynomial time the optimal solution of PSE. BestOrS uses the OrS property. We denote by $C(x, r)$ the cost of an optimal solution of problem PSE when there are r machines and tasks $1, \ldots, x$ ($x \in \{1, \ldots, n\}$, and $r \in \{1, \ldots, m\}$). When extending an allocation from r to $r + 1$ machines, BestOrS checks allocations with $1, 2, \ldots, (x-1)$ smallest tasks on machine r (we recall that the tasks are indexed in non-increasing order of loads: $p_1 \geq p_2 \geq \cdots \geq p_n$). More formally, for all $x \in \{1, \ldots, n\}$, and $r \in \{1, \ldots, m-1\}$, we have:

$$C(x, r+1) = \min_{i \in \{1,\ldots,x-1\}} \left(C(x-i, r) + i \sum_{j=x-i+1}^{x} p_j \right). \tag{5}$$

The cost of an allocation on a single machine can be directly computed. For each $x \in \{1, \ldots, n\}$, we have:

$$C(x, 1) = x \sum_{i=1}^{x} p_i. \tag{6}$$

The minimum cost of a solution of PSE is $C(n, m)$. By backtracking, we can deduce from $C(n, m)$ an allocation of minimum cost (for example, when each value $C(x, r)$ is computed, we record the tasks which are on the r-th machine).

Proposition 5.1. *Algorithm* BestOrS *computes in* $O(n^2 m)$ *an optimal allocation of problem* PSE *when there is a single type.*

Proof. Let us first show that for each $x \in \{1, \ldots, n\}$ and $r \in \{1, \ldots, m\}$, the value $C(x, r)$ computed by BestOrS is the minimum cost to allocate the x largest tasks on r machines. Once we have shown this, we can deduce that $C(n, m)$ is the minimum cost of a solution of PSE, and thus that this algorithm returns an optimal solution.

The proof is by induction on r, the number of machines. When there is only one machine, there is only one possible allocation, and its cost is equal to the number of tasks times the load of the machine. Thus the cost is given by Equation (6).

Let us now assume that for each $y \in \{1, \ldots, x-1\}$, $C(y, r)$ is the minimum cost to allocate the y largest tasks of the instance on r machines. Let us show that $C(x, r+1)$ is the minimum cost to allocate the x largest tasks of the instance on $r + 1$ machines. By Lemma 4.1, there exists an optimal OrS allocation. Thus, there exists an optimal allocation O of the x largest tasks on $r + 1$ machines, where the smallest tasks are on the same machine. In O, let $i^* \in \{1, \ldots, n\}$ be the number of tasks which are on the machine to which the smallest task x is allocated. As O is an OrS allocation, this machine has the i^* smallest tasks $x - i^* + 1, \ldots, x$. The cost of O is $C(x - i^*, r) + i^* \sum_{j=x-i^*+1}^{x} p_j$. Indeed, $\sum_{j=x-i^*+1}^{x} p_j$ is the cost of each of the i^* smallest tasks in O and $C(x - i^*, r)$ is by induction the minimum cost to allocate the other tasks (the $x - i^*$ largest tasks on r machines). Equation (5) computes the cost of a feasible solution. If $i' \in \{1, \ldots, n\}$ is the value of i that minimizes $C(x - i, r) + i \sum_{j=x-i+1}^{x} p_j$ then Equation (5) computes the cost of a solution where the i' smallest tasks are on the same machine, and the other tasks are partitioned

optimally on the r remaining machines. This value is minimized when $i' = i^*$. Thus $C(x, r+1)$ is the minimum cost of a partition of the x largest tasks on $r+1$ machines.

Therefore, $C(n, m)$ is the cost of an optimal solution of PSE. We now show that $C(n, m)$ can be be computed in $O(n^2 m)$. We store the values $C(x, r)$ on an $n \times m$ matrix. Each value $C(x, r+1)$ can be computed in $O(n)$ once the values $C(x - i, r)$ are known. For each $x \in \{2, \ldots, n\}$, $\sum_{j=2}^{x} p_j$ can be computed at the beginning of the algorithm in $O(n)$. Then, when we compute $C(x, r+1)$ we have $x \leq n$ costs to examine—each cost being computed in $O(1)$ if we start by $i = x - 1$ and decrement i until $i = 1$. □

6. Several types

In this section we propose a series of algorithms solving the Partition with Side Effects (PSE) problem when there are several types. We start with a dynamic programming algorithm polynomial in the number of tasks and machines, but exponential in the number of admissible sizes of tasks and in the number of types. Then, using the OrS property (derived in Section 4), we propose an algorithm, called CutJuxtapose (Section 6.3), polynomial in the number of tasks, but exponential in the number of types and machines. Finally, we show that BestOrS (the algorithm proposed for a single type in Section 5) is an approximation algorithm for PSE.

Before analyzing the general PSE problem, we mention several special cases that can be optimally solved using the results presented in Section 5.

Independent types: When for all $i \neq j$ $\alpha_{i,j} = 0$, the types are independent. Separately for each type, we use BestOrS to assign the task of this type to m machines. We obtain an optimal solution in $O(Tn^2 m)$.

Equivalent types: If there is a value C such that for each pair of types t and t' (including $t = t'$), $\alpha_{t,t'} = C$, then all the types are equivalent. The impact of one task on another does not depend on its type. By using BestOrS once to allocate all tasks on all machines, we obtain an optimal solution in $O(n^2 m)$.

Large influences: If influences are very large for each pair of types $t' \neq t$ (i.e., when $\alpha_{t',t} > \sum_{t \in \mathcal{T}} (n^t \alpha_{t,t} \sum_{i \in J^{t'}} p_i)$), and if $T \leq m$, then sharing machines between tasks of different types is inefficient. $\sum_{t \in \mathcal{T}} n^t \alpha_{t,t} \sum_{i \in J^{t'}} p_i$ is the cost of a *dedicated* partition that allocates all tasks of each type to one of T dedicated machines (types do not share machines). When the influences are large, if a task shares a machine with a different type, its cost is larger than the total cost of a dedicated partition.

We call a *configuration* a mapping from types to the number of machines assigned to each type, such that the total number of assigned machines is m. Given a configuration, we can compute an optimal partition in polynomial time by using T times the $O(n^2 m)$-algorithm of Section 5. Thus, an optimal partition is the partition of the minimum cost over all the configurations. In order to count the possible configurations, we show how to generate them. Since there is at least one machine per type, without loss of generality we assign machine $m - i + 2$ to type i, for each $i \in \{2, \ldots, T\}$ (one machine will be assigned to type 1 later). We assign the remaining $m - T + 1$ machines as follows. We pick $(T - 1)$ numbers $\{k_1, \ldots, k_{T-1}\}$ such that $1 \leq k_1 \leq k_2 \leq \cdots \leq k_{T-1} \leq m - T + 1$. To the first type, we assign machines $[1, k_1]$. To the second type, if $k_2 > k_1$, we assign machines $[k_1 + 1, k_2]$; otherwise ($k_2 = k_1$), type 2 has no machines assigned in this phase. We continue with the remaining but the last type. The last type T is assigned the remaining machines $[k_{T-1} + 1, m - T + 1]$. The number of ways to pick numbers $\{k_1, \ldots, k_{T-1}\}$ from the set $\{1, \ldots, m - T + 1\}$ is equal to the number of combinations with repeti-

tion, $\binom{(m-T+1)+(T-2)}{T-1} = \binom{m-1}{T-1}$. Thus the complexity of our algorithm is $O\left(\binom{m-1}{T-1} n^2 m\right) \subset O(n^2 m^T)$. This is a polynomial time algorithm when T is a constant.

6.1. Allocation for a fixed number of sizes

We present a dynamic programming algorithm, DynSize, which solves PSE in polynomial time if the number of types is constant and if the number of possible sizes for the tasks is also constant.

If the number of possible sizes of each task is fixed, $p_i \in \mathbb{P}$, where $\mathbb{P} \subset \mathbb{N}$ is the set of admissible sizes and $|\mathbb{P}|$ is a constant (does not depend on the instance). For each type $t \in \mathcal{T}$, we denote by l_t the number of different sizes of a task of type t ($l_t \leq |\mathbb{P}|$). We write p_t^j for the j-th size of a task of type t, where $j \in \{1, \ldots, l_t\}$. Note that t in subscript distinguishes p_t^j from p_j^t, the size of the j-th largest task of type t. We denote by C_t^j the set of the tasks which are of type t and of size p_t^j. We put $n_t^j = |C_t^j|$.

By $C(y_1^1, \ldots, y_1^{l_1}, y_2^1, \ldots, y_2^{l_2}, \ldots, y_T^1, \ldots, y_T^{l_T}, r)$ we denote the cost of an optimal solution of PSE when there are r machines and y_t^j tasks of type t and of size p_t^j. We will use a shorthand notation $C\left((y_t^j), r\right)$ (with $t \in \{1, \ldots, T\}$ and $j \in \{1, \ldots, l_t\}$). We prove in Proposition 6.1 that the following dynamic programming algorithm, DynSize, finds in polynomial time an optimal solution of PSE.

The cost of an optimal allocation on a single machine is:

$$C\left((y_t^j), 1\right) = \sum_{t=1}^{T} \sum_{t'=1}^{T} \alpha_{t,t'} \left(\sum_{k=1}^{l_t} y_t^k p_t^k \right) \left(\sum_{k=1}^{l_{t'}} y_{t'}^k \right). \tag{7}$$

The cost of an optimal allocation on $r \geq 2$ machines is:

$$C\left((y_t^j), r\right) = \min_{(x_t^j : x_t^j \in \{0, \ldots, y_t^j\})} \left(C\left((y_t^j - x_t^j), r-1\right) + C\left((x_t^j), 1\right) \right). \tag{8}$$

The cost of an optimal solution of PSE is $C((n_t^j), m)$. We can deduce from $C(n, m)$ an allocation of minimum cost by backtracking (for example, when each value $C(x, r)$ is computed, we record the tasks which are on the r-th machine).

Proposition 6.1. DynSize *optimally solves* PSE *in* $O(mn^{2\sum_{t \in \mathcal{T}} l_t})$.

Proof. When there is a single machine, there is only one possible allocation. Its cost is the sum over all the pairs of types $(t, t') \in \mathcal{T}^2$ of the cost that the tasks of type t imply on the cost of the tasks of type t'. This cost is expressed by the right hand side of Equation (7). The value $C\left((y_t^j), 1\right)$ is thus valid.

For each $t \in \{1, \ldots, T\}$, and then for each $j \in \{1, \ldots, l_t\}$, let $y_t^j \in \{0, \ldots, n_t^j\}$ be a number of tasks of type t and of size p_t^j, and let $x_t^j \in \{0, \ldots, y_t^j\}$. The expression $C\left((y_t^j - x_t^j), r-1\right) + C\left((x_t^j), 1\right)$ computes the cost of an optimal solution among the solutions where there are x_t^j tasks of type t and of size p_t^j on machine M_r, and where there are $y_t^j - x_t^j$ tasks of type t and of size p_t^j on machines M_1 to M_{r-1}. In any partition of the tasks (y_t^j), the number of tasks of type t and of size p_t^j on M_r is between 0 and y_t^j. Thus, the right hand side of Equation (8) computes the minimum

cost of a partition where there are y_t^j tasks of type t and of size p_t^j to assign to r machines, and Equation (8) is valid.

Therefore, by using Equations (7) and (8), we can compute $C\left((n_t^j), m\right)$, the cost of an optimal solution of problem PSE. Let us now analyze the time complexity of this algorithm.

On a given subset of the machines, the number of tasks of type t and size p_t^j is between 0 and n_t^j. Thus, the number of possible vectors (y_t^j) is $\prod_{t\in\mathcal{T}, j\in\{1,\dots,l_t\}}(n_t^j + 1) < (n+1)^{\sum_{t\in\mathcal{T}} l_t}$. Thus, the number of possible values of $C\left((y_t^j), r\right)$ to compute is smaller than $m(n+1)^{\sum_{t\in\mathcal{T}} l_t}$. Each value $C\left((y_t^j), 1\right)$ is computed in $O(T^2(\max_t l_t)^2) \subset O(T^2n^2)$. To compute $C\left((y_t^j), r\right)$ with $r \geq 2$, we use stocked vectors $C\left((y_t^j), r-1\right)$. The minimum in Equation (8) checks all valid (non-negative) vectors $(y_t^j - x_t^j)$. The number of such vectors is at most equal to the number of possible vectors (y_t^j), i.e., $(n+1)^{\sum_{t\in\mathcal{T}} l_t}$. Therefore the complexity of this dynamic programming algorithm is $O(mn^2 \sum_{t\in\mathcal{T}} l_t)$. □

Corollary 6.1. *If the number of types and the number of possible sizes for the tasks are constant, the above described dynamic programming algorithm optimally solves* PSE *in polynomial time.*

6.2. Approximation scheme

In this section, we use DynSize, the dynamic programming algorithm introduced in the previous section, to derive an approximation scheme, ApproxScheme. ApproxScheme has a low complexity if both the number of types and the size of the largest task are small.

ApproxScheme is defined as follows. Let $\varepsilon > 0$. Let I be an instance of PSE, and let $p_{\max} = \max_{i\in\{1,\dots,n\}} p_i$ be the size of the largest task of I. First, construct a new instance I' by rounding the size of each job to $p'_i = (1+\varepsilon)^{\lceil \log_{(1+\varepsilon)}(p_i)\rceil}$. Then, run DynSize (Section 6.1) on I', and output the given allocation.

Proposition 6.2. *Let $\varepsilon > 0$, and let I be an instance of* PSE. *Algorithm* ApproxScheme(I, ε) *computes in $O(mn^{2T\lceil \log_{1+\varepsilon}(p_{\max})\rceil})$ a $(1+\varepsilon)$-approximate solution for* PSE.

Proof. Let us show that this algorithm returns a $(1+\varepsilon)$-approximate solution for PSE. Let P be an assignment (partition) of the tasks of I to the machines. We denote by $C(P)$ the cost of solution P when the tasks have their real sizes (the sizes given in I). We denote by $C'(P)$ the cost of solution P on instance I'. Let O be a partition of the tasks of I which is optimal for PSE for instance I, and let O' be a partition which is optimal for I'.

Since O' is an optimal solution for I', we have

$$C'(O') \leq C'(O) \tag{9}$$

At the rounding step, the size of each task is increased by a factor of at most $1+\varepsilon$. Therefore, the cost of each task is increased by a factor of at most $1+\varepsilon$. Therefore, for each partition P, we have:

$$C(P) \leq C'(P) \leq (1+\varepsilon)C(P) \tag{10}$$

Using Inequalities (9) and (10), we get:

$$C(O') \leq C'(O') \leq C'(O) \leq (1+\varepsilon)C(O) \tag{11}$$

Since the dynamic programming algorithm introduced in Section 6.1 is an exact algorithm, the solution given by ApproxScheme(I, ε) is O'. This solution is thus a $(1+\varepsilon)$-approximate solution of PSE on instance I.

Let us now show that this algorithm runs in $O(mn^{2T\lceil\log_{1+\varepsilon}(p_{\max})\rceil})$. The rounding step is done in $O(n)$. At the end of this step, there are at most $\lceil\log_{1+\varepsilon}(p_{\max})\rceil$ sizes. Thus, for each type t, there are at most $\lceil\log_{1+\varepsilon}(p_{\max})\rceil$ different sizes of tasks of type t in I'. In I', $\sum_{t\in\mathcal{T}} l_t \leq T\lceil\log_{1+\varepsilon}(p_{\max})\rceil$. The running time of the dynamic programming algorithm on I' is thus $O(n^{2T\lceil\log_{1+\varepsilon}(p_{\max})\rceil}m)$. □

Although the complexity of ApproxScheme may be exponential if $p_{\max}$ is very high, its complexity is much lower than the one of the dynamic programming algorithm of Section 6.1. If T and $p_{\max}$ are constants, then the number of possible sizes is a constant (it is at most $Tp_{\max}$), and thus the dynamic programming algorithm is a polynomial time algorithm. However, its complexity is huge ($O(mn^{2Tp_{\max}})$), making it impractical. On the contrary, ApproxScheme, which is also a polynomial time algorithm in this case, has a complexity of $O(mn^{2T\lceil\log_{1+\varepsilon}(p_{\max})\rceil})$

If an instance has many small tasks and a constant number n_l of large tasks, in order to reduce the complexity of the algorithm, it is better to round only the small tasks. Let p_x be the size of the largest among the small tasks. We will obtain $\lceil\log_{(1+\varepsilon)}(p_x)\rceil$ rounded sizes. Thus, in the dynamic programming algorithm of Section 6.1, we will get vectors of size $n_l + T\lceil\log_{(1+\varepsilon)}(p_x)\rceil$ instead of $T\lceil\log_{(1+\varepsilon)}(p_{\max})\rceil$.

We will give in Section 6.3 an exact algorithm, called CutJuxtapose, with complexity of $O(n^{(m-1)T}(m!)^{T-1})$. If m is large, as it is often the case, this algorithm cannot be used in practice. On the contrary, the complexity of ApproxScheme increases linearly in m. With a large number of machines, small number of types and relatively homogeneous tasks' sizes, ApproxScheme returns acceptable solutions faster than CutJuxtapose.

6.3. CutJuxtapose: partition using the OrS *property*

We show in this section an optimal algorithm for PSE. This algorithm, called CutJuxtapose, uses the OrS property (Section 4). CutJuxtapose is exponential in the number of types T and in the number of machines m, but polynomial in the number of tasks n.

The OrS property specifies, for each type, an optimal ordering of tasks (from the largest to the smallest). Independently for each type t, CutJuxtapose cuts the ordered sequence of tasks into a set of at most m sub-sequences. A sub-sequence corresponds to tasks that will be assigned to the same machine; two sub-sequences of the same type will be assigned to different machines. Then, CutJuxtapose juxtaposes (combines) sets corresponding to tasks of different types. In the first phase, CutJuxtapose generates all possible cuts. Then, for each cut, CutJuxtapose tests all possible combinations of juxtaposing sub-sequences. Therefore, CutJuxtapose examines all OrS assignments and returns the optimal one.

Proposition 6.3. *Algorithm* CutJuxtapose *computes in* $O(n^{(m-1)T}(m!)^{T-1})$ *an optimal allocation of* PSE.

Proof. By Lemma 4.1 there exists an optimal partition which fulfills the OrS property. Algorithm CutJuxtapose considers all possible OrS partitions (cuts and combinations), thus it finds the one with the minimal cost.

In order to determine the number of partitions examined by CutJuxtapose, let us first observe that the number of ways to split a sequence S of length n into at most k subsequences $S_1, \ldots, S_k$ is at most $\binom{n+k-1}{k-1}$. One way to show this is the following. Put $n + k - 1$ squares in a sequence. Remove $k-1$ squares to leave n squares and at most $k-1$ gaps. We now have at most k sequences of squares. We then match the sequence S to the remaining sequences of squares. The matching

splits S into at most k subsequences. There are $\binom{n+k-1}{k-1}$ ways to remove the squares and so at most $\binom{n+k-1}{k-1}$ ways to split the sequence.

Algorithm CutJuxtapose considers the partitions which fulfill the OrS property, which determines the order of the tasks. Therefore, $n^{(i)}$ tasks of type i form a sequence that is cut into at most m subsequences (each subsequence corresponds to tasks allocated to the same machine). Thus there are at most $\binom{n^{(i)}+m-2}{m-1}$ cuts to examine.

Given a most m subsequences for each type, the number of ways to juxtapose these subsequences is at most $(m!)^{T-1}$ since all the permutations of the subsequences of type $t \neq 1$ are computed (without loss of generality we assume that the i^{th} sequence of type 1 is on machine i). Thus the time complexity of CutJuxtapose is $\left(\prod_{i\in\{1,\dots,T\}} \binom{n^{(i)}+m-2}{m-1}\right)(m!)^{T-1}$, which is in $O(n^{(m-1)T}(m!)^{T-1})$. □

If the number of machines, m, and the number of types, T, are constant, then the complexity of CutJuxtapose is $O(n^{T(m-1)})$: it is a polynomial time algorithm.

When there are only two types A and B, juxtaposing reduces to finding the minimum cost bipartite matching in a bipartite graph $(\{J_1^A,\dots,J_m^A\},\{J_1^B,\dots,J_m^B\})$ (i.e. each J_i^A has an edge with each J_j^B). Let $L_i^A = \sum_{j\in J_i^A} p_j$ be the load of J_i^A, and let L_j^B be the load of J_i^B. Let M be the maximum value of $|J_i^A|\alpha_{B,A}L_j^B + |J_j^B|\alpha_{A,B}L_j^A$ $(i,j \in \{1,\dots,m\}^2)$. The cost of matching J_i^A with J_j^B is equal to $M - (|J_i^A|\alpha_{B,A}L_j^B + |J_j^B|\alpha_{A,B}L_j^A)$. By solving bipartite matching with the Kuhn–Munkres algorithm [Edmonds and Karp, 1972], the complexity of juxtapose phase is $O(m^3)$, and thus the complexity of the whole CutJuxtapose is $O(n^{2(m-1)}m^3)$.

6.4. BestOrS *as an approximation algorithm*

We demonstrate in this section that BestOrS, the algorithm we proposed for a single type, is an approximation algorithm for the general case.

Proposition 6.4. *Let* $\alpha_{\max} = \max_{(t,t')\in\mathcal{T}^2} \alpha_{t,t'}$*, and* $\alpha_{\min} = \min_{(t,t')\in\mathcal{T}^2} \alpha_{t,t'}$*. Algorithm* BestOrS *is an* $\frac{\alpha_{\max}}{\alpha_{\min}}$*-approximate algorithm for* PSE*, and the bound* $\frac{\alpha_{\max}}{\alpha_{\min}}$ *is asymptotically tight.*

Proof. Let us consider an instance I of PSE, and let O_I be an optimal solution of I, of cost $C(O_I)$. Let I' be the instance obtained from I by keeping the same tasks (number and lengths) but by replacing all the $\alpha_{t,t'}$ by $\alpha_{\min}$. Let O'_I be an optimal solution of I', of cost $C(O_{I'})$. We have: $C(O_I) \geq C(O_{I'})$. If, in $O_{I'}$, we replace all the $\alpha_{\min}$ by the original $\alpha_{t,t'}$, then the cost of the solution is increased by at most $\frac{\alpha_{\max}}{\alpha_{\min}}$ (the cost of each task is multiplied by at most $\frac{\alpha_{\max}}{\alpha_{\min}}$). The solution $O_{I'}$ can be obtained by algorithm BestOrS (all the values $\alpha_{t,t'}$ are equal in this solution). Since the cost of this solution is at most $\frac{\alpha_{\max}}{\alpha_{\min}}$ times the minimum cost, $C(O_I)$, BestOrS is an $\frac{\alpha_{\max}}{\alpha_{\min}}$-approximate algorithm.

In order to show that the bound is asymptotically tight, let us consider the following instance. There are m machines and m types. There are m tasks of each type. All the tasks are of size 1. For each type t, $\alpha_{t,t} = \alpha_{\min}$, and for each pair of types t and t' such that $t \neq t'$, $\alpha_{t,t'} = \alpha_{\max} \geq \alpha_{\min}$. The optimal solution has cost $m^3\alpha_{\min}$. In this partition, there is a machine for each type—all the tasks of the same type are on the same machine. Each of the m^2 tasks has a cost $m\alpha_{\min}$. BestOrS can return the partition where on each machine there is one task of each type. The cost of each task is then $(m-1)\alpha_{\max} + \alpha_{\min}$, and the cost of the partition is $m^2((m-1)\alpha_{\max} + \alpha_{\min})$. The approximation ratio of BestOrS is thus at least $\frac{m^2((m-1)\alpha_{\max}+\alpha_{\min})}{m^3\alpha_{\min}} = \frac{m-1}{m}\frac{\alpha_{\max}}{\alpha_{\min}} + \frac{1}{m}$. This tends towards $\frac{\alpha_{\max}}{\alpha_{\min}}$ when m tends towards infinity. □

7. Extension: convex costs

In this section, we study general cost functions, and we focus on the case where there is a single type. So far, we have studied a linear cost function: if all the tasks have the same type the cost of each task is proportional to the load of its machine. However, more complex cost functions are interesting from the systems perspective (e.g., webserver's response time as a function of load is convex [Cao et al., 2003, Khanna et al., 2006, Slothouber, 1996]). We show in Section 7.1 that if the cost function is strictly convex then problem PSE becomes strongly NP-hard. A concave cost function is not as realistic, since it would mean that, with unit size tasks, the average cost of a task decreases when the number of tasks on the same machine increases.

More formally, we assume in this section that the cost of task i in partition P is $c_i = f(L_{M_{P,i}})$, where f is a strictly convex function (and $L_{M_{P,i}}$ is the load of the machine on which task i is allocated in P, $L_{M_{P,i}} = \sum_j$ on machine $_{M_{P,i}}$ p_j). Let us denote by PCSE (which stands for Partition with Convex Side Effects) the following problem (note that this problem has a natural extension to several types):
Input: n tasks (of different sizes), a number m of machines, and an increasing and strictly convex cost function f.
Output: a partition which minimizes the sum of the costs $\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} f(L_{M_{P,i}})$.

7.1. Complexity for strictly convex cost

In this section, we show that it is NP-hard to minimize costs given by any strictly convex cost function, even for a single type.

Proposition 7.1. *For any cost function f which is increasing and strictly convex, the decision version of* PCSE *is strongly NP-complete.*

Proof. We reduce from the strongly NP-complete 3-Partition [Garey and Johnson, 1979]. An instance of the 3-Partition consists of a finite set A of $3q$ elements, a bound $B \in \mathbb{Z}^+$, and a size $s(a) \in \mathbb{Z}^+$ for each $a \in A$, such that $s(a)$ satisfies $\frac{B}{4} < s(a) < \frac{B}{2}$ and such that $\sum_{a \in A} s(a) = qB$. The question is: can A be partitioned into q disjoint sets $S_1, \ldots, S_q$ such that, for $i \in \{1, \ldots, q\}$, $\sum_{a \in S_i} s(a) = B$?

We study the decison version of PCSE, which is as follows: given an instance of PCSE, is there a partition P with total cost $C(P) = \sum_{i=1}^{n} c_i$ at most K?

From an instance of 3-Partition we construct an instance of PCSE as follows. We have $m = q$ machines $\{M_1, \ldots, M_m\}$ and $n = 3q$ tasks. For each element $a \in A$, we have a task j_a of size $p_a = s(a)$. We set the budget $K = nf(B)$.

Let us now show that there is a solution to the 3-Partition if and only if there is a solution to the corresponding instance of PCSE. Assume first that there is a solution to the 3-Partition. Let $\{S_1, \ldots, S_m\}$ be the sets of the solution. Consider a partition P in which tasks corresponding to the elements in S_i are allocated on M_i. In P, since $\sum_{a \in S_i} s(a) = B$, the load on each machine is equal to B. The cost c_i of each task is thus $f(B)$, and the sum of the costs is $nf(B)$. Consequently, there is a solution of cost $nf(B) = K$ for PCSE.

Let us now assume that there is a solution P to PCSE. We show that there is a solution to the corresponding instance of the 3-Partition problem. Let n_i be the number of tasks on M_i in P, and let L_i be the load of M_i in P. The cost of P is $C(P) = \sum_{i=1}^{m} n_i f(L_i)$.

We will show that since the cost of the solution P is at most K, then on each machine (1) there are exactly 3 tasks; and (2) the load is exactly B.

In order to obtain a contradiction, suppose that in P there is a machine M_i such that $n_i > 3$. If $n_i > 3$ then $L_i > B$, since there are at least 4 tasks on M_i and each task is strictly larger than $\frac{B}{4}$. If there is a machine with $n_i > 3$, then there exists another machine M_j with $n_j < 3$. Since all the tasks are smaller than $\frac{B}{2}$, we have $L_j < B$.

Hence,

$$n_i f(L_i) + n_j f(L_j) > (n_i - 1)f(L_i) + (n_j + 1)f(L_j)$$

This argument can be used until $n_k = 3$, for each machine M_k. Therefore, we get $C(P) = \sum_{k=1}^{m} n_k f(L_k) > \sum_{k=1}^{m} 3f(L_k)$. Since f is strictly convex,

$$\sum_{k=1}^{m} 3f(L_k) \geq 3mf\left(\frac{\sum_{k=1}^{m} L_k}{m}\right) = 3mf(B) = K.$$

Thus, $C(P) > K$, which leads to a contradiction. There are thus exactly three tasks per machine.

We now show that the load of each machine is exactly B. Since there are exactly three tasks per machine, $C(P) = 3\sum_{i=1}^{m} f(L_i)$. Since P is a solution to PCSE, $C(P) \leq K = 3mf(B) = 3mf\Big((\sum_{k=1}^{m} L_k)/m\Big)$. We thus have $\sum_{i=1}^{m} f(L_i) \leq mf\Big((\sum_{k=1}^{m} L_k)/m\Big)$. As f is strictly convex, the inequality holds only if $L_i = (\sum_{k=1}^{m} L_k)/m$ for each machine M_i.

Therefore, in P on each machine there are exactly 3 tasks and the load is B. If we denote by S_i the set of elements of A corresponding to the tasks allocated on M_i in P, we have for each set S_i, with $i \in \{1, \ldots, m\}$, $\sum_{a \in S_i} s(a) = B$. This defines a solution to the 3-Partition problem.

There is a solution to the 3-Partition if and only if there is a solution to the corresponding instance of the PCSE. As PCSE is (trivially) in NP, PCSE is thus strongly NP-complete.

□

Proposition 7.2. *For any cost function f which is increasing and strictly convex, the decision version of* PCSE *is NP-complete, even if there are only 2 machines.*

Proof. We do a reduction from problem EqualPartition, which is NP-complete, to our problem. The EqualPartition problem is the following one: the input is a finite set A of $2k$ elements. Each $a \in A$ has size $s(a) \in \mathbb{N}$. Let $S = \sum_{a \in A} s(a)$. The question is: can A be partitioned into two disjoint sets A_1, A_2 of equal cardinality k such that $\sum_{a \in A_1} s(a) = \sum_{a \in A_2} s(a) = S/2$? EqualPartition is an NP-complete variant of the NP-complete Partition problem [Garey and Johnson, 1979] (the anwer for an instance I of Partition is the answer of the instance of EqualPartition in which we have the tasks of I and an equal number of artificial tasks of size 0).

The instance of PCSE corresponding to an instance of the EqualPartition is as follows. We have 2 machines $\{M_1, M_2\}$ and $n = 2k$ tasks. For each element $a \in A$, we introduce a task j_a of size $p_a = 2S + s(a)$. We set the budget $K = 2kf\left(2kS + \frac{S}{2}\right)$.

Let us now show that there is a solution to the EqualPartition problem if and only if there is a solution to the corresponding instance of the PCSE problem. Assume first that there is a solution A_1, A_2 to the EqualPartition problem. Let us consider the partition where, for each $i \in \{1, 2\}$, the tasks which correspond to the elements of A_i are allocated to M_i. In this partition, since there are k elements in A_i and since $\sum_{a \in A_i} s(a) = \frac{S}{2}$, the load on each machine is equal to $2kS + \frac{S}{2}$. The cost of each task is thus $f(2kS + \frac{S}{2})$, and the sum of the costs is $2kf(2kS + \frac{S}{2}) = K$.

Let us now assume that there is a solution P to the PCSE problem, and let us show that there is a solution to the corresponding instance of the EqualPartition problem. Let n_j be the number of tasks on M_j in P, and let L_j be the load of M_j in P. The cost of P is $C(P) = n_1 f(L_1) + n_2 f(L_2)$.

First, we show by contradiction that the number of tasks on each machine is k. Assume that there is in P a machine M_i such that $n_i > k$. Assume without loss of generality that $M_i = M_1$. In this case, on the other machine, M_2, there are at most $k - 1$ tasks.

On M_1, the load is $L_1 \geq (k+1)2S$, since there are at least $k+1$ tasks and each task has a size at least $2S$. Pick a task on M_1 of size $2S + r$ (where $r \geq 0$). The load on M_2 is $L_2 \leq (k-1)2S + (S-r)$, since there are at most $(k-1)$ tasks, each of size $2S + s(a)$, and $\sum_{a \text{ on } M_2} s(a) \leq (S - r)$. If this task of size $2S + r$ is moved to M_2 then the load on M_1 becomes $L'_1 \geq k2S$ (there are still at least k tasks of size at least $2S$). On M_2, the load after the move is $L'_2 \leq (k-1)2S + (S-r) + (2S + r)$. Thus $L'_2 \leq k2S + S < L_1$. After the move, the cost of each of the $n_1 - 1 \geq k$ tasks remaining on M_1 is decreased from $f(L_1)$ to $f(L'_1)$. The cost of each of the $n_2 + 1 \leq k$ tasks on M_2 is increased by $f(L'_2) - f(L_2) < f(L_1) - f(L'_1)$ (the last inequality follows from f being strictly convex). Therefore, moving a task from M_1 to M_2 decreases the total cost, which contradicts the assumption that P is optimal.

Therefore, in an optimal solution of PCSE there are exactly k tasks per machine. We now analyze the cost of an optimal solution. Assume that the load on M_1 is $2kS + \Delta$, and the load on M_2 is $2kS + \Delta'$ (with $\Delta + \Delta' = S$). The cost of the solution is then $kf(2kS + \Delta) + kf(2kS + \Delta')$. Since f is strictly convex, the cost is minimized when $\Delta = \Delta' = \frac{S}{2}$. Therefore, if we denote by A_i the set of elements of A corresponding to the tasks allocated on M_i in P, we have for each set A_i, with $i \in \{1, \ldots, m\}$, $\sum_{a \in S_i} s(a) = \frac{S}{2}$. There is thus a solution to the EqualPartition problem.

There is a solution to the EqualPartition if and only if there is a solution to the corresponding instance of the PCSE. As PCSE is (trivially) in NP, PCSE is thus NP-complete. □

7.2. An approximation algorithm for convex costs

Since problem PCSE is strongly NP-complete even with only one type, there is no polynomial time algorithm to solve it unless $P = NP$. We analyze the approximation ratio of algorithm BestOrS, which considers an input of PCSE as an input of PSE with only one type (i.e. the instance of PSE is are the tasks with sizes equal to ones from PCSE). Let W be the total load in the system: $W = \sum_{i=1}^{n} p_i$. Let $\Delta = \frac{f(W)}{W}$. We assume that $f(1) = 1$ (we can scale the instance such that this is true).

Proposition 7.3. *Algorithm* BestOrS *is a* Δ*-approximate algorithm for* PCSE.

Proof. Let I be an instance of PCSE. Let P be a partition of the tasks of I on the machines. We denote by $C_{\text{PCSE}}(P)$ the cost of this partition for problem PCSE. We denote by $C_{\text{PSE}}(P)$ be the cost of this partition for problem PSE (in this case the tasks—number and lengths— are the same as in the instance of PCSE but the convex cost function has been replaced by a linear cost function).

Given a partition P, the cost of task i for problem PSE is $C_{\text{PSE}}(i) = L_{M_{P,i}}$, where $L_{M_{P,i}}$ is the load of the tasks on the same machine than i in P. Given this same partition, the cost of task i for problem PCSE is $C_{\text{PCSE}}(i) = f(L_{M_{P,i}})$. Since f is an increasing convex function, and since $L_{M_{P,i}} \leq W$, we have $\frac{f(L_{M_{P,i}})}{L_{M_{P,i}}} \leq \frac{f(W)}{W} = \Delta$. Thus $f(L_{M_{P,i}}) \leq \Delta L_{M_{P,i}}$. We have:

$$\begin{aligned}\frac{C_{\text{PCSE}}(P)}{C_{\text{PSE}}(P)} &= \frac{\sum_{i=1}^{n} C_{\text{PCSE}}(i)}{\sum_{i=1}^{n} C_{\text{PSE}}(i)} \\ &= \frac{\sum_{i=1}^{n} f(L_{M_{P,i}})}{\sum_{i=1}^{n} L_{M_{P,i}}} \\ &\leq \frac{\sum_{i=1}^{n} \Delta L_{M_{P,i}}}{\sum_{i=1}^{n} L_{M_{P,i}}} \leq \Delta\end{aligned} \tag{12}$$

Let OPT_{PCSE} and OPT_{PSE} be optimal solutions of PCSE and PSE on instance I. Since OPT_{PSE} is an optimal solution of problem PSE, its cost is smaller than or equal to the costs of all the other partitions: in particular we have $C_{\text{PSE}}(\text{OPT}_{\text{PSE}}) \leq C_{\text{PSE}}(\text{OPT}_{\text{PCSE}})$. Furthermore, we have $C_{\text{PSE}}(\text{OPT}_{\text{PCSE}}) \leq C_{\text{PCSE}}(\text{OPT}_{\text{PCSE}})$ since f is an increasing convex function and $f(1) = 1$. Using these two inequalities and Equation (12), we have:

$$\begin{aligned}C_{\text{PSE}}c(\text{OPT}_{\text{PSE}}) &\leq \Delta\, C_{\text{PSE}}(\text{OPT}_{\text{PSE}}) \\ &\leq \Delta\, C_{\text{PSE}}(\text{OPT}_{\text{PCSE}}) \\ &\leq \Delta\, C_{\text{PCSE}}(\text{OPT}_{\text{PCSE}})\end{aligned} \tag{13}$$

Therefore, algorithm BestOrS, which returns solution OPT_{PSE}, is a Δ-approximate algorithm for PCSE.

□

This algorithm can be used if the cost function is close to be linear (i.e. if Δ is close to 1). However, if the cost function is convex and increases quickly (i.e. if $f(x+1) \gg f(x)$), then the maximum cost of a task is likely to be minimized. In this case, an optimal algorithm (or an approximation scheme) for the widely studied scheduling problem which consists in minimizing the makespan on parallel machines, $P||C_{\max}$, can be useful. Indeed, using such an algorithm on the instance of PCSE (n tasks of size $\{p_1, \ldots, p_n\}$) is likely to return a good solution (assignment of the tasks to the machines) for PCSE. Note that, in the general case, an optimal algorithm for $P||C_{\max}$ is n-approximate for PCSE. Indeed, let $C^*_{\max}$ be the makespan of an optimal solution of $P||C_{\max}$. In an optimal solution of PCSE, the maximum load is at least $C^*_{\max}$ and thus the cost of an optimal solution of PCSE is at least $f(C^*_{\max})$. In an optimal solution of $P||C_{\max}$, the load on each machine is at most $C^*_{\max}$. The cost of such a solution is thus at most $nf(C^*_{\max})$: it is a n-approximate solution.

8. Experiments

To experimentally verify the performance of the proposed algorithms, we conducted simulations on a dataset derived from a data center trace published by Google [Reiss et al., 2012]. We first describe our methodology: the algorithms we use, our method of converting the Google trace to a series of instances of our problem, and two methods we use to normalize the results of our algorithms. Then, we describe the results of the experiments.

8.1. Method

8.1.1. Algorithms

We compared CutJuxtapose with the following algorithms.

BestOrS (Section 6.4) converts the instance with multiple types to an instance with a single type (maintaining task sizes), and then runs the dynamic programming algorithm for a single type. Then, the resulting assignment is converted back to a multi-type solution (types are assigned to tasks in order of type indices).

Hill performs hill-climbing starting from an initial assignment of all tasks to machines. Given an initial assignment, for each type, the algorithm tries all possible movements of tasks between machines that maintain the OrS order (analyzing them one by one). If no improvement is possible, the algorithm stops. Otherwise, the algorithm accepts allocation having the lowest overall cost and, in the next iteration, tries to improve it. For each type, the algorithm maintains a list of machines ordered by the size of the largest task assigned to that machine, and a set of free machines (to which no task of the type is assigned). The following moves are considered: migrating the smallest task to the previous (in the OrS order) machine; migrating the largest task to the next machine; migrating the smallest task to any free machine; migrating the largest task to any free machine.

We used various initial assignments for Hill. BestOrS+hill starts with the allocation returned by BestOrS. one+hill starts with all tasks assigned to a single machine. dedic+hill starts from an allocation that spreads types on available machines—all tasks of type i are allocated on machine $i \mod m$.

These algorithms can have large runtimes. In particular, CutJuxtapose is exponential in the number of machines and types; and hill-climbing can test potentially all possible allocations. During our experiments, we limited the runtime of each algorithm by limiting to $N = 10^6$ the total number of allocations the algorithm can check. If this limit is reached, the algorithm returns the best assignment found so far. Hill tests thus at most N moves. In CutJuxtapose, the limit translates to the number of evaluations of the cost of a tested partition (in the juxtapose phase). If the limit is reached, the algorithm stops.

8.1.2. Data

We used the Google Cluster Trace [Reiss et al., 2012] as an input data. The trace describes all tasks running during a month on one of the Google clusters. For each task, the trace reports in its task record table, among other data, the task's CPU, memory and disk IO usage averaged over a 5-minute long period. This trace is certainly not ideal for our needs: the trace reports the usage of raw resources (CPU, memory, network, disk), and not the load of applications. However, to the best of our knowledge, there are no publicly-available traces describing loads and performance of applications (in contrast to raw resources).

We generated a random sample of $10,000$ task records. Each task record corresponds to a task in our model. To generate loads and types, we use data on the mean CPU utilization and the assigned memory. We normalize CPU and memory utilization by dividing each by its maximum. We remove 45% of task records that reported less than 0.005 in both normalized CPU and memory usage. The traces also provide disk IO usage. We do not use it, as for only 3% of the tasks the normalized disk usage was higher than both normalized CPU and normalized memory usage. Thus, a great majority of tasks have negligible disk IO usage.

We assign types using ratio ρ of normalized CPU to normalized memory usage. Figure 1 shows a histogram of ρ on our dataset. $\rho = 1$ (in Figure 1, a dashed line at $\log(\rho) = 0$) partitions

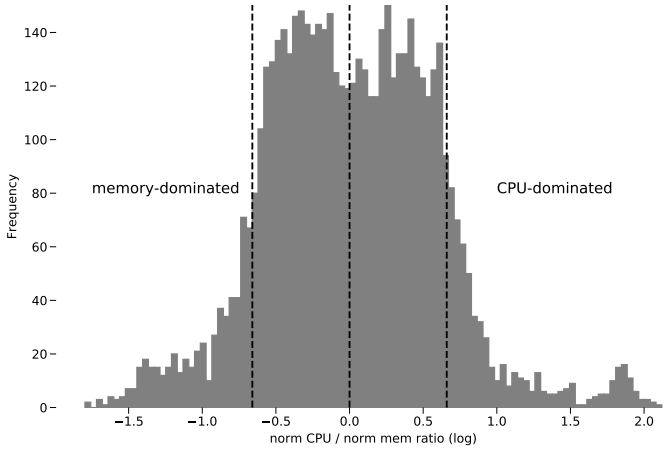

Figure 1: Ratio of CPU to memory usage in our 10k dataset (histogram of the logarithm of the ratio). Dashed lines denote boundaries we used to partition the dataset into up to 4 types.

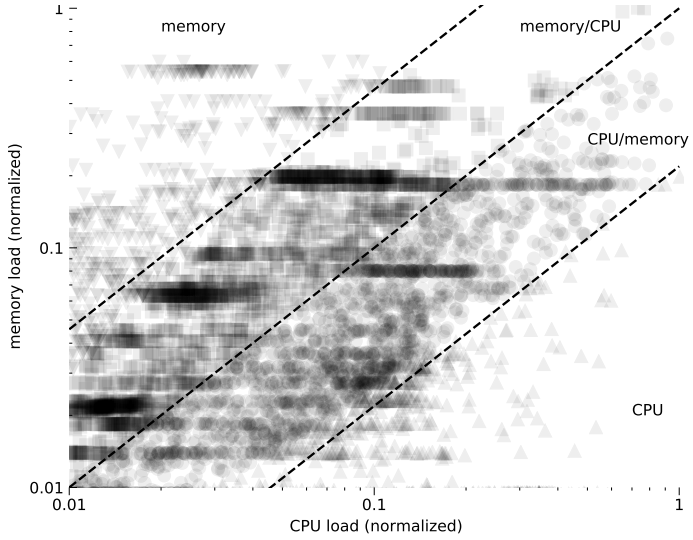

Figure 2: Task loads and types derived from the 10k dataset. Axes denote the CPU and memory usage from the dataset (normalized to the maximum usage). The load of a task is the maximum from the point's coordinates. Four symbols (the triangle, the circle, the square and the inverted triangle) denote to which type a task belongs in instances with four types. Dashed lines denote boundaries for types. In instances with two types the diagonal defines the boundary between types (for example the CPU type contains tasks denoted by circles and triangles). In instances with three types the diagonal is not used.

the dataset in almost equal halves. We use two other tresholds, $\log(\rho) = -0.66$ and $\log(\rho) = 0.66$. They distinguish tasks for which usage of one resouce clearly dominates the other: 10% of the tasks are memory-dominated (with $\log(\rho) < -0.66$), while 11% of the tasks are CPU-dominated (with $\log(\rho) > 0.66$).

To assign types, we use the tresholds defined above. For $T = 2$, we use treshold $\rho_1 = 1$. For $T = 3$, we use thresholds $\log(\rho_1) = -0.66$ and $\log(\rho_2) = 0.66$. For $T = 4$ we use thresholds $\log(\rho_1) = -0.66$, $\rho_2 = 1$, and $\log(\rho_3) = 0.66$.

We set the coefficients $\alpha_{t,t'}$ based on how compatible are the resource requirements. After the conversion process described above, t_1 is a type for which memory dominates CPU, while the for last type, t_T, CPU dominates the memory. Thus, the further apart the type numbers, the more compatible these types are. Coefficients are symmetric ($\alpha_{t,t'} = \alpha_{t',t}$) and normalized so that $\alpha_{t,t} = 1$. Then, for two types ($T = 2$), $\alpha_{1,2} = \alpha_{2,1} = 0.5$; for three types ($T = 3$), $\alpha_{1,2} = \alpha_{2,3} = 0.5$, while $\alpha_{1,3} = 0.25$; finally, for four types ($T = 4$), each $t, t' = t + 1$ have $\alpha_{t,t'} = 0.75$, each $t, t' = t + 2$ have $\alpha_{t,t'} = 0.5$, and $\alpha_{1,4} = 0.25$.

To assign load to a task, we take the maximum from the weighted CPU and weighted memory, multiply this maximum by 100 and round to the nearest integer. Figure 2 shows an overview of the resulting distribution of loads by task types. We label type 1 as memory (10%), type 2 as memory-CPU (40%), type 3 as CPU-memory (39%), and type 4 as CPU (11%).

To generate an instance of n tasks belonging to T types, we take a random sample of n tasks from the dataset; thus, the proportions of types in the generated instance are similar to the dataset. However, a random sample might have less than T types: if this is the case, we remove a task from the most common type in the instance and add a task of the missing type.

Overall, we generate instances with the number of types $T \in \{2, 3, 4\}$, the number of tasks $n \in \{10, 20, 50, 100\}$ and the number of machines $m \in \{2, 3, 5, 10\}$. For each combination of n, T and m, we generate 30 instances. We have 1440 instances in total.

8.1.3. Presentation of results

We show the cost $C(P)$ of the partition returned by the algorithms normalized to (i.e., divided by) the cost of the optimal partition returned by CutJuxtapose. However, as CutJuxtapose is exponential in the number of types and machines, it does not complete in reasonable time on larger instances. We thus derive a lower bound on the minimum cost as the cost of the partition returned by BestOrS on an instance with tasks having the same lengths but belonging to a single type t^* with $\alpha_{t^*,t^*} = \min_{(t,t')} \alpha_{t,t'}$.

8.2. Results

CutJuxtapose, the exact algorithm, is exponential in the number of types and machines, and thus computationally expensive in larger instances. We limited the number of tested assignments to one million, which resulted in limiting the running time by roughly one minute on our computer. Overall, CutJuxtapose solved 464 instances (out of 1440 tested). CutJuxtapose failed to solve all instances with $m = 10$ machines and all but one instance with $m = 5$ machines. Similarly, for $n = 100$ tasks, CutJuxtapose solved only the smallest instances with two machines and two and three types.

Next, we analyzed the quality of the solutions returned by various heuristics on these 464 instances in which CutJuxtapose finished (as in these instances we had the cost of the optimal solution). Figure 3 shows the results (we use boxplots in which the middle line shows the median and the box spans between 25th and 75th quantile). BestOrS+hill, hill climbing starting from

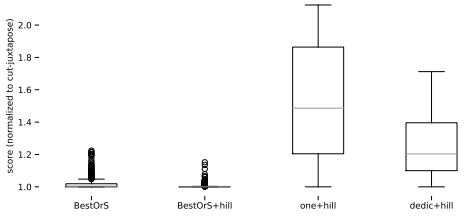

Figure 3: The total cost $C(P)$ of the solutions returned by various heuristics normalized to the total cost of the solution returned by CutJuxtapose. 464 instances, in which CutJuxtapose finished in less than 10^6 evaluations.

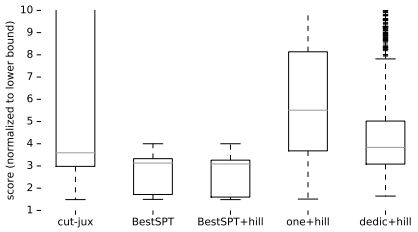

Figure 4: The total cost $C(P)$ of the solutions returned by various heuristics and by CutJuxtapose. All algorithms limited to 10^6 evaluations. The cost normalized to the lower bound. All instances.

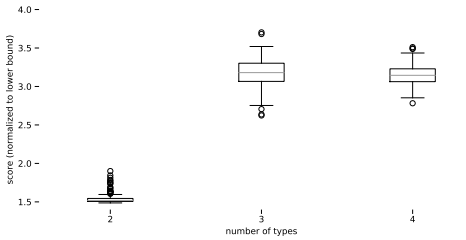

Figure 5: The total cost $C(P)$ of the solution returned by CutJuxtapose. $C(P)$ normalized to the lower bound. 464 instances, in which CutJuxtapose finished in less than 10^6 evaluations.

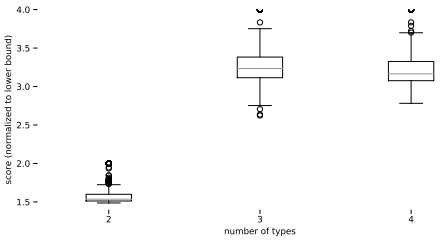

Figure 6: The total cost $C(P)$ of the solution returned by BestOrS+hill, hill climbing starting from the dynamic programming solution. The cost normalized to the lower bound. All instances.

Table 1: Percentage of instances that CutJuxtapose finished analyzing in 10^6 evaluations; CutJuxtapose finished a single instance with $m = 5$ and didn't finish any instance $m = 10$.

	$n \rightarrow$	10	20	50	100
m	T				
2	2	100	100	100	100
	3	100	100	100	100
	4	100	100	77	0
3	2	100	100	0	0
	3	100	67	0	0
	4	100	0	0	0

the dynamic programming algorithm solution performs best, finding the optimal solution in 68% of the cases (compared to 32% for the dynamic programming without hill climbing). The cost of the worst solution of BESTORS+HILL was within 15% of the cost of the optimum.

To analyze the quality of the solutions on all instances, we start with the analysis of the tightness of the proposed lower bound. Figure 5 shows the cost of the optimal solutions found by CUTJUXTAPOSE (on 464 instances in which CUTJUXTAPOSE finished) normalized by the lower bound (defined in Section 8.1.3). As CUTJUXTAPOSE is optimal, if the lower bound was tight, these normalized cost should be roughly 1. The average score for two types is 1.54, while for three types is 3.16, and for four types is 3.15. The main reason for the increased normalized score is the reduction of the cost of the lower bound. The lower bound uses the minimum coefficient $\alpha_{t,t'}$. For two types the minimum coefficient is 0.5, while for three and four types the minimum coefficient is 0.25.

Figure 4 shows the cost of the solution returned by all heuristics on all instances (costs normalized to the lower bound). On all instances, CUTJUXTAPOSE performs worst, as in 10^6 evaluations it is not able to find the right area of the (huge) search space (note that if CUTJUXTAPOSE is interrupted, it returns the minimum cost partition found so far). BESTORS+HILL performs best, but only slightly better than the dynamic programming solution.

Figure 6 shows a detailed breakdown of BESTORS+HILL performance as a function of the number of types. By comparing these results to the results of CUTJUXTAPOSE (Figure 5), we see that BESTORS+HILL has a performance similar to the performance of CUTJUXTAPOSE on smaller instances: BESTORS+HILL average scores for 2, 3 and 4 types are 1.59, 3.27 and 3.23 (compared to CUTJUXTAPOSE scores of 1.54, 3.16 and 3.15, respectively).

Moreover, BESTORS+HILL converges quickly. On the average, the heuristics tests 185 configurations (standard deviation 345; the maximum number of configurations tested is 2028).

9. Related work

Alternative models of data center resource management. There is no standard model of data center resource management (standard in the sense in which the parallel job model is standard for HPC). Existing models can be roughly categorized into variants of multi-dimensional bin-packing (to model heterogeneous resource requirements), stochastic optimization (to model uncertainty), and statistical approaches. Other issues include pricing and revenue management [Pschel et al., 2015] (in our model, the provider does not select jobs to execute); or distributed scheduling [Sebastio et al., 2017] (our model targets a single data center, rather than a distributed cloud).

Multi-dimensional bin-packing. In bin-packing approaches, tasks are modeled as items to be packed into bins (machines) of known capacity [Coffman Jr et al., 1996, Delorme et al., 2016]. For instance [Gullhav et al., 2017] considers the problem of packing replicated services onto VMs having various capacities. To model heterogeneous tasks and resources, bin packing is extended to vector packing: an item's size is defined as a vector with dimensions corresponding to requirements on individual resources (CPU, memory, disk or network bandwidth) [Stillwell et al., 2012]. These are hard optimization problems: bin packing is strongly NP-hard (but has an asymptotic polynomial time approximation scheme, PTAS [Fernandez de la Vega and Lueker, 1981]), while two-dimensional vector packing does not admit an asymptotic PTAS [Woeginger, 1997] Alternatively, if tasks have unit-size requirements, simpler representations can be used, such as maximum weighted matching [Beaumont et al., 2013]. However, we claim that the bin packing is too imprecise to capture performance of tasks in a data center. First, some

tasks may have better performance when executing on machines with smaller loads [Slothouber, 1996], while the bin packing model implicitly assumes that as long as a bin is not overloaded, tasks' performance is the same. Second, bin packing does not permit even small overpacking, while data center resource managers commonly oversubscribe at least for CPU [Reiss et al., 2012], leading to probabilistic service level agreements (SLAs, see also stochastic approaches below). Our cost function permits us to model the gradually worsening performance in function of the overpacking degree. Third, bin packing ignores performance degeneration resulting from colocated tasks competing for the shared physical resources, such as processor's cache or the bandwidth between the processor and the memory [Kambadur et al., 2012, Kim et al., 2015, Koh et al., 2007, Podzimek et al., 2015, Xu et al., 2013].

Stochastic versions of combinatorial optimization problems. Stochastic versions of classic optimization problems [Chen et al., 2011, Goel and Indyk, 1999, Wang et al., 2011] can be used to model uncertainty of tasks' resource requirements or their variability in time. In these representations, some parameters of an instance are random variables, e.g., items' sizes in bin packing. A typical goal is to construct an optimal solution (in terms of the number of bins used, or the value of the items picked to a knapsack) that violates the capacity constraints only with a small probability. These models, however, rarely lead to practical algorithms, at the same time requiring restrictive assumptions on the stochastic models of jobs, as usually the algorithms work only for a certain distribution.

Statistical approaches. [Bobroff et al., 2007] use statistical information on the past CPU load of tasks (CDF, autocorrelation, periodograms) to predict the load in the following time period; then they use bin packing to calculate a partition minimizing the number of used bins subject to a constraint on the probability of overloading servers. [Di et al., 2015] analyze resource sharing for streams of tasks to be processed by virtual machines. Sequential and parallel task streams are considered in two scenarios—when there are sufficient resources to run all tasks, and when the resources are insufficient. For sufficient resources, optimality conditions are formulated. For insufficient resources, fair scheduling policies are proposed.

Analysis of effects of colocation. [Podzimek et al., 2015] analyze the performance of colocated CPU-intensive tasks. Their measured performance interference metric is similar to our α_{t_j,t_i} coefficient. [Kim et al., 2015] focuses on experimental measures of performance interference between a few concrete HPC applications. This interference, called the affinity metric, is similar to our α_{t_j,t_i} coefficients. They propose a greedy allocation heuristics, but they neither study the complexity of the problem nor demonstrate the optimality of the solutions found by their heuristics.

Game-theoretic approaches. There is a strong connection between our model and games, in which each task is owned by a selfish agent who wants to minimize task's cost.

Load balancing games. Our model relates to the load-balancing games introduced by [Koutsoupias and Papadimitriou, 1999], in which the cost of each task is the total load of the machine to which the task is allocated. This model represents, e.g., a system of servers from which users download large files: tasks correspond to requests of individual users and each the user aims at contacting a server with the smallest load [Vöcking, 2007]. This model is analyzed also for unrelated machines [Azar et al., 2015]. Contrarily to what we do in this paper, the game model considers that each task is owned by an agent, and that each agent chooses the machine on which its task will be scheduled. In most papers, the authors aim at minimizing the maximum load over all the machines (see [Vöcking, 2007] for a survey), but in some papers [Awerbuch et al., 2005, Christodoulou and Koutsoupias, 2005] the aim is to minimize the average social cost, as we do in

our paper. However, to the best of our knowledge, no centralized optimal algorithm to minimize the average cost of the tasks has been studied. In Section 5, we have given a polynomial time algorithm which solves this problem.

Coalition structure generation. Our model is also related to the coalition structure generation (CSG, see [Elkind et al., 2013] for a recent overview). CSG consists of the following. The set of agents (tasks) is partitioned into subsets (called coalitions). Each agent affects the cost of the coalition she is assigned to (but not the costs of the other coalitions). However, in CSG the aim is to minimize the total cost of all coalitions (and not the average cost of an agent). Additionally, in CSG the number of coalitions is not bounded, while we bound the number of subsets by the number of machines m. [Aziz and De Keijzer, 2011] analyze CSG with players having types. When the number of types is a constant, they give a polynomial algorithm. However, their notion of type is more restrictive than ours: two players have the same type if their influence on the costs of the others is exactly the same.

10. Discussion and conclusions

We propose a new model describing the performance of tasks colocated on machines. Our model introduces the notion of *type*. Types describe and allow to deal with tasks' heterogeneity: e.g., a computationally-intensive task influences the performance of a webserver in a different way than a database instance. In our model, a task influences other tasks as a function of its size *and its type*.

In this paper (except in Section 7), we consider a linear cost function. Linear costs roughly correspond to classic optimality measures while non-linear costs model complex end-user performance. In linear cost function, the cost of task i is the sum of the *weighted* sizes of all tasks j assigned to machine M. The weight (corresponding to the coefficient α_{t_i,t_j}), which depends on i's and j's types, measures the compatibility between the two types. Large weights correspond to types that compete for similar resources. Small weights correspond to types that complement each other, e.g., a CPU-intensive and a memory-intensive tasks.

Our model has three main advantages. First, it captures tasks' heterogeneity. Second, it may optimize the observed (experienced) performance of the tasks, and not just the usage of the resources. Third, it is a minimal extension of a standard scheduling model. Tasks' affinities or interferences, similar to our notion of type, were proposed in recent systems papers on colocation performance [Kim et al., 2015, Podzimek et al., 2015]. In contrast to bin-packing models [Stillwell et al., 2012], we do not use a strict limit on machines' capacities. Hardware resources are limited, but a task does not abruptly fail when, e.g., the total processor usage (or the disk IO, bandwidth or even memory, with OS swapping) gets to 100%. Instead, tasks' performance is gradually degraded, resulting in slower observed response times. Additionally, some tasks (e.g. webservers [Slothouber, 1996]) have better observed performance if the total processor usage is 20–40%, rather than 90–100%. The aim of our model is to go beyond the crisp constraints of bin-packing, which unrealistically treats any packing not exceeding the capacity as equally good, while not permitting even small overpacking.

The notion of type in our model should be expressive enough to describe tasks with heterogeneous resource requirements. We show a series of polynomial algorithms. BestOrS (Section 5) is polynomial in the number of machines and tasks but applicable to instances with a single type, or with types that are independent, equivalent, or strictly incompatible (Section 6). DynSize (Section 6.1) is polynomial in the number of machines and tasks but requires a fixed number of

tasks' sizes and is exponential in the number of types. ApproxScheme (Section 6.2) uses DynSize for instances with small number of tasks' sizes. Finally, CutJuxtapose is polynomial in the number of tasks, but exponential in the number of machines and types (Section 6.3). The key to efficiency is thus to define a limited number of types—at most logarithmic in the number of tasks—as our results show that the problem is strongly NP-complete if we allow for arbitrary number of types (Section 7.1).

Our experimental results (Section 8) demonstrate that, while CutJuxtapose is feasible for only few types and machines, BestOrS+hill performs well. BestOrS+hill is a standard local search algorithm starting from a reasonable configuration and greedily improving it until hitting a local optimum. Its good performance might suggest that our problem has a regular combinatorial structure, and thus more complex meta-heuristics (such as taboo search) might further improve the results.

A general convex cost function, instead of just optimizing the load of the machine, might model the observed performance of a task, such as the commonly-used 95th percentile response time for user-facing services. Such a function might thus encapsulate the response-time models [Cao et al., 2003, Khanna et al., 2006, Slothouber, 1996]. The function might be even derived from measurements of actual response times of tasks under various loads and various colocation scenarios. The problem is strongly NP-complete with convex costs (Section 7.1). However, the complexity stems from the cost function, not the notion of type (as the problem is NP-complete even for a single type). We also show an approximation algorithm for convex costs (Section 7.2).

We leave open the NP-completeness of Partition with Side Effects with a constant number of types.

We are currently working on validating our model in a real data center resource manager, which would open many interesting questions on, e.g., automatic classification of tasks into types or inferring their coefficients.

In this paper, we minimize the total cost. Another natural research direction is to add weights to the tasks (and to minimize the weighted total cost), or to minimize the maximum cost. Note that even for a single type and two machines this last problem is NP-hard since it reduces to the widely studied scheduling problem of minimizing the makespan on parallel machines ($P||C_{\max}$).

The notion of type of a task can also be applied to generalize other problems, as for example load balancing games (with agents choosing on which machines their tasks will be scheduled). Similarly, types may be also used to reduce the complexity of coalition structure generation problems.

Acknowledgements. We thank the reviewers for their careful reading of this paper and their constructive remarks and suggestions. In particular, we thank Reviewer 1 for the time he or she took to provide exceptionally detailed remarks, which greatly improved the quality of our paper. This research has been partly supported by the Polish National Science Center grant Sonata (UMO-2012/07/D/ST6/02440), and a Polonium grant (joint programme of the French Ministry of Foreign Affairs, the Ministry of Science and Higher Education and the Polish Ministry of Science and Higher Education).

Agarwal, A., Charikar, M., Makarychev, K., Makarychev, Y., 2005. O($\sqrt{log\ n}$) approximation algorithms for Min UnCut, Min 2CNF deletion, and directed cut problems. In: ACM Symposium on Theory of Computing (STOC), Proc. ACM, pp. 573–581.

Awerbuch, B., Azar, Y., Epstein, A., 2005. The price of routing unsplittable flow. In: ACM Symposium on Theory of Computing (STOC), Proc. pp. 57–66.

Azar, Y., Fleischer, L., Jain, K., Mirrokni, V., Svitkina, Z., 2015. Optimal coordination mechanisms for unrelated machine scheduling. Operations Research 63 (3), 489–500.

Aziz, H., De Keijzer, B., 2011. Complexity of coalition structure generation. In: International Conference on Autonomous Agents and Multiagent Systems (AAMAS), Proc. pp. 191–198.

Beaumont, O., Eyraud-Dubois, L., Thraves Caro, C., Rejeb, H., 2013. Heterogeneous resource allocation under degree constraints. IEEE Transactions on Parallel and Distributed Systems 24 (5), 926–937.

Bobroff, N., Kochut, A., Beaty, K., 2007. Dynamic placement of virtual machines for managing SLA violations. In: IFIP/IEEE International Symposium on Integrated Network Management (IM), Proc. IEEE, pp. 119–128.

Cao, J., Andersson, M., Nyberg, C., Kihl, M., 2003. Web server performance modeling using an M/G/1/K* PS queue. In: International Conference on Telecommunications (ICT), Proc. Vol. 2. IEEE, pp. 1501–1506.

Chen, M., Zhang, H., Su, Y.-Y., Wang, X., Jiang, G., Yoshihira, K., 2011. Effective vm sizing in virtualized data centers. In: IFIP/IEEE International Symposium on Integrated Network Management (IM), Proc. IEEE, pp. 594–601.

Christodoulou, G., Koutsoupias, E., 2005. The price of anarchy of finite congestion games. In: ACM Symposium on Theory of Computing (STOC), Proc. pp. 67–73.

Coffman Jr, E. G., Garey, M. R., Johnson, D. S., 1996. Approximation algorithms for bin packing: A survey. In: Hochbaum, D. (Ed.), Approximation algorithms for NP-hard problems. PWS, pp. 46–93.

Creignou, N., Khanna, S., Sudan, M., 2001. Complexity Classifications of Boolean Constraint Satisfaction Problems. SIAM.

Delorme, M., Iori, M., Martello, S., 2016. Bin packing and cutting stock problems: Mathematical models and exact algorithms. European Journal of Operational Research 255 (1), 1–20.

Di, S., Kondo, D., Cappello, F., 2014. Characterizing and modeling cloud applications/jobs on a Google data center. The Journal of Supercomputing 69 (1), 139–160.

Di, S., Kondo, D., Wang, C., 2015. Optimization of composite cloud service processing with virtual machines. IEEE Transactions on Computers 64, 1755–1768.

Edmonds, J., Karp, R. M., 1972. Theoretical improvements in algorithmic efficiency for network flow problems. Journal of the ACM 19 (2), 248–264.

Elkind, E., Rahwan, T., Jennings, N. R., 2013. Computational coalition formation. In: Weiss, G. (Ed.), Multiagent Systems. MIT Press, pp. 329–380.

Fernandez de la Vega, W., Lueker, G., 1981. Bin packing can be solved within 1 + epsilon in linear time. Combinatorica 1 (4), 349–355.

Garey, M. R., Johnson, D. S., 1979. Computers and Intractability: A Guide to the Theory of NP-Completeness.

Garg, N., Vazirani, V. V., Yannakakis, M., 1993. Approximate max-flow min-(multi)cut theorems and their applications. In: ACM Symposium on Theory of Computing (STOC), Proc. ACM, pp. 698–707.

Goel, A., Indyk, P., 1999. Stochastic load balancing and related problems. In: IEEE Annual Symposium on Foundations of Computer Science (FOCS). IEEE, pp. 579–586.

Gullhav, A. N., Cordeau, J.-F., Hvattum, L. M., Nygreen, B., 2017. Adaptive large neighborhood search heuristics for multi-tier service deployment problems in clouds. European Journal of Operational Research 259, 829–846.

Kambadur, M., Moseley, T., Hank, R., Kim, M. A., 2012. Measuring interference between live datacenter applications. In: International Conference on High Performance Computing, Networking, Storage and Analysis (SC), Proc. IEEE, p. 51.

Khanna, G., Beaty, K., Kar, G., Kochut, A., 2006. Application performance management in virtualized server environments. In: Network Operations and Management Symposium, (NOMS), Proc. IEEE, pp. 373–381.

Kim, S., Hwang, E., Yoo, T.-K., Kim, J.-S., Hwang, S., Choi, Y.-R., 2015. Platform and co-runner affinities for many-task applications in distributed computing platforms. In: IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), Proc. IEEE CS, pp. 667–676.

Klusácek, D., Rudová, H., 2014. Multi-resource aware fairsharing for heterogeneous systems. In: JSSPP, Proc.

Koh, Y., Knauerhase, R., Brett, P., Bowman, M., Wen, Z., Pu, C., 2007. An analysis of performance interference effects in virtual environments. In: IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS), Proc. IEEE, pp. 200–209.

Koutsoupias, E., Papadimitriou, C., 1999. Worst-case equilibria. In: Annual Conference on Theoretical Aspects of Computer Science (STACS), Proc. Springer, pp. 404–413.

Podzimek, A., Bulej, L., Chen, L. Y., Binder, W., Tuma, P., 2015. Analyzing the impact of cpu pinning and partial cpu loads on performance and energy efficiency. In: IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), Proc. IEEE CS, pp. 1–10.

Pschel, T., Schryen, G., Hristova, D., Neumann, D., 2015. Revenue management for Cloud computing providers: Decision models for service admission control under non-probabilistic uncertainty. European Journal of Operational Research 244 (2), 637–647.

Reiss, C., Tumanov, A., Ganger, G. R., Katz, R. H., Kozuch, M. A., 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In: ACM Symposium on Cloud Computing (SoCC), Proc. ACM, p. 7.

Sebastio, S., Gnecco, G., Bemporad, A., 2017. Optimal distributed task scheduling in volunteer clouds. Computers & Operations Research 81, 231–246.

Slothouber, L. P., 1996. A model of web server performance. In: International World Wide Web Conference (WWW).
Stillwell, M., Vivien, F., Casanova, H., 2012. Virtual machine resource allocation for service hosting on heterogeneous distributed platforms. In: International Parallel and Distributed Processing Symposium (IPDPS), Proc. IEEE, pp. 786–797.
Vöcking, B., 2007. Selfish load balancing. In: Nisan, N., Roughgarden, T., Tardos, E., Vazirani, V. V. (Eds.), Algorithmic Game Theory. Cambridge, pp. 517–542.
Wang, M., Meng, X., Zhang, L., 2011. Consolidating virtual machines with dynamic bandwidth demand in data centers. In: INFOCOM, Proc. IEEE, pp. 71–75.
Woeginger, G., 1997. There is no asymptotic PTAS for two-dimensional vector packing. Information Processing Letters 64 (6), 293–297.
Xu, Y., Musgrave, Z., Noble, B., Bailey, M., 2013. Bobtail: Avoiding long tails in the cloud. In: USENIX conference on Networked Systems Design and Implementation (NSDI), Proc. pp. 329–341.